Introduction: World of microcontrollers

The situation we find ourselves today in the field of microcontrollers had its beginnings in the development of technology of integrated circuits. This development has enabled us to store hundreds of thousands of transistors into one chip. That was a precondition for the manufacture of microprocessors. The first computers were made by adding external peripherals such as memory, input/output lines, timers and others to it. Further increasing of package density resulted in creating an integrated circuit which contained both processor and peripherals. That is how the first chip containing a microcomputer later known as a microcontroller has developed.

This is how it all got started...

In the year 1969, a team of Japanese engineers from BUSICOM came to the USA with a request that a few integrated circuits for calculators were to be designed according to their projects. The request was sent to INTEL and Marcian Hoff was in charge of the project there. Having experience working with a computer, the PDP8, he came up with an idea to suggest fundamentally different solutions instead of the suggested design. This solution presumed that the operation of integrated circuit was to be determined by the program stored in the circuit itself. It meant that configuration would be simpler, but it would require far more memory than the project proposed by Japanese engineers. After a while, even though the Japanese engineers were trying to find an easier solution, Marcian's idea won and the first microprocessor was born. A major help with turning an idea into a ready-to-use product was Federico Faggin. Nine months after hiring him, Intel succeeded in developing such a product from its original concept. In 1971 Intel obtained the right to sell this integrated circuit. Before that Intel bought the license from BUSICOM which had no idea what a treasure it had. During that year, a microprocessor called the 4004 appeared on the market. That was the first 4-bit microprocessor with the speed of 6000 operations per second. Not long after that, an American company CTC requested from Intel and Texas Instruments to manufacture an 8-bit microprocessor to be applied in terminals. Even though CTC gave up this project, Intel and Texas Instruments kept working on the microprocessor and in April 1972 the first 8-bit microprocessor called the 8008 appeared on the market. It was able to address 16Kb of memory, had 45 instructions and the speed of 300 000 operations per second. That microprocessor was the predecessor

of all today's microprocessors. Intel kept on developing it and in April 1974 it launched an 8-bit processor called the 8080. It was able to address 64Kb of memory, had 75 instructions and initial price was $360.

Another American company called Motorola, quickly realized what was going on, so they launched 8-bit microprocessor 6800. Their chief constructor was Chuck Peddle. Apart from the processor itself, Motorola was the first company that also manufactured other peripherals such as the 6820 and 6850. At that time many companies recognized the greater importance of microprocessors and began their own development. Chuck Peddle left Motorola to join MOS Technology and kept working intensively on developing microprocessors.

At the WESCON exhibition in the USA in 1975, a crucial event in the history of the microprocessors took place. MOS Technology announced that it was selling processors 6501 and 6502 at $25 each, that interested customers could purchase immediately. It was such a sensation that many thought it was a kind of fraud, considering that competing companies were selling the 8080 and 6800 at $179 each. On the first day of the exhibit, in response to the competitor, both Motorola and Intel cut the prices of their microprocessors to $69.95. Motorola accused MOS Technology and Chuck Peddle of plagiarizing the protected 6800. Because of that, MOS Technology gave up further manufacture of the 6501, but kept manufacturing the 6502. It was the 8-bit microprocessor with 56 instructions and ability to directly address 64Kb of memory. Due to low price, 6502 became very popular so it was installed into computers such as KIM-1, Apple I, Apple II, Atari, Commodore, Acorn, Oric, Galeb, Orao, Ultra and many others. Soon several companies began manufacturing the 6502 (Rockwell, Sznertek, GTE, NCR, Ricoh, Commodore took over MOS Technology). In the year of its prosperity 1982, this processor was being sold at a rate of 15 million processors per year!

Other companies did not want to give up either. Frederico Faggin left Intel and started his own company Zilog Inc. In 1976 Zilog announced the Z80. When designing this microprocessor Faggin made a crucial decision. The 8080 had already been developed and he realized that many would remain loyal to that processor because of the great expenditures which rewriting of all the programs would result in. Accordingly he decided that a new processor had to be compatible with the 8080, i.e. it had to be able to perform all the programs written for the 8080. Apart from that, many other features have been added so that the Z80 was the most powerful microprocessor at that time. It was able to directly address 64Kb of memory, had 176 instructions, a large number of registers, a built-in option for refreshing dynamic RAM memory, a single power supply, greater operating speed etc. The Z80 was a great success and everybody replaced the 8080 by the Z80. Certainly the Z80 was commercially the most successful 8-bit microprocessor at that time. Besides Zilog, other new manufacturers such as Mostek, NEC, SHARP and SGS appeared soon. The Z80 was the heart of many computers such as: Spectrum, Partner, TRS703, Z-3 and Galaxy.

In 1976 Intel came up with an upgraded version of the 8-bit microprocessor called the 8085. However, the Z80 was so much better that Intel lost the battle. Even though a few more microprocessors appeared later on the market (6809, 2650, SC/MP etc.), the die had already been cast. There were no such great improvements which could make manufacturers to change their mind, so the 6502 and Z80 along with the 6800 remained chief representatives of the 8-bit microprocessors of that time.

Microcontroller versus Microprocessor

A microcontroller differs from a microprocessor in many ways. The first and most important difference is its functionality. In order that the microprocessor may be used, other components such as memory must be added to it. Even though the microprocessors are considered to be powerful computing machines, their weak point is that they are not adjusted to communicating to peripheral equipment.

Simply, In order to communicate with peripheral environment, the microprocessor must use specialized circuits added as external chips. In short microprocessors are the pure heart of the computers. This is how it was in the beginning and remains the same today.

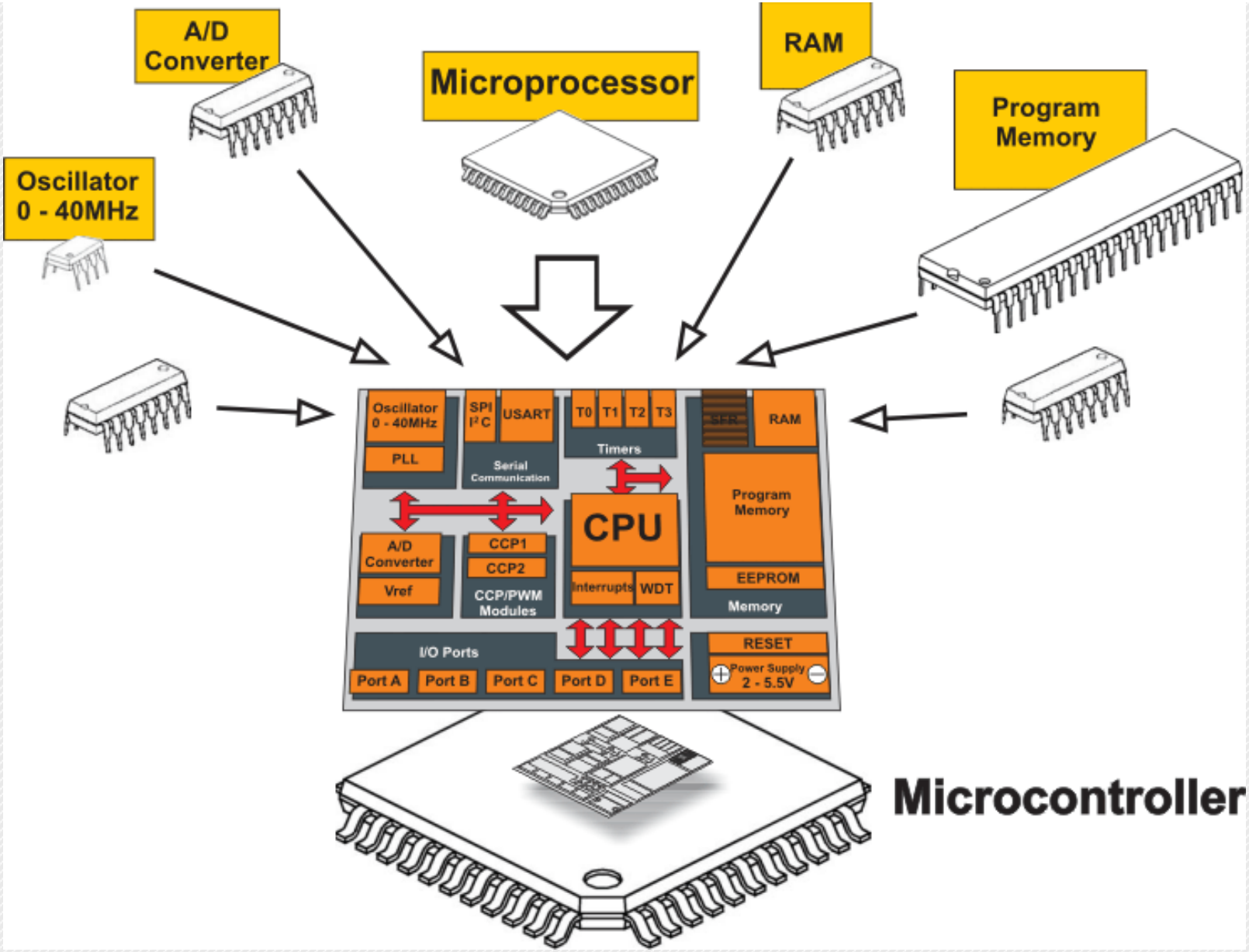


Fig. 0-1 Microcontroller versus Microprocessor

On the other hand, the microcontroller is designed to be all of that in one. No other specialized external components are needed for its application because all necessary circuits which otherwise belong to peripherals are already built into it. It saves the time and space needed to design a device.

BASIC CONCEPT

Did you know that all people can be classified into one of 10 groups- those who are familiar with binary number system and those who are not familiar with it. You don't understand? That means that you still belong to the later group. If you want to change your status read the following text describing briefly some of the basic concepts used further in this book (just to be sure we are on the same page).

World of Numbers

Mathematics is such a good science! Everything is so logical and simple as that. The whole universe can be described with ten digits only. But, does it really have to be like that? Do we need exactly ten digits? Of course not, it is only a matter of habit. Remember the lessons from the school. For example, what does the number 764 mean: four units, six tens and seven hundreds. Simple! Could it be described in a bit more complicated way? Of course it could: 4 + 60 + 700. Even more complicated? Naturally: 4*1 + 6*10 + 7*100. Could this number look a bit more scientific? The answer is yes: 4*10^0 + 6*10^1 + 7*10^2. What does it actually mean? Why do we use exactly these numbers: 100, 101 and 102 ? Why is it always about the number 10? That is because we use ten different digits (0, 1, 2, ... 8, 9). In other words, because we use base-10 number system, i.e. decimal number system.

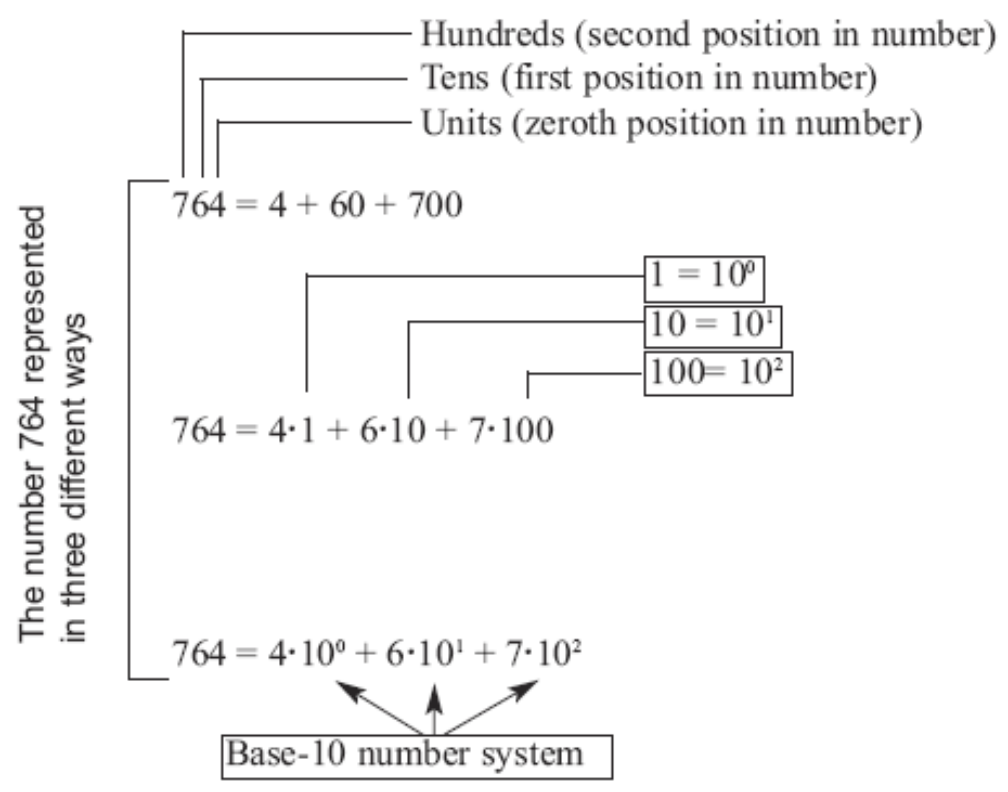


Fig. 0-2 The number 764 represented in three different ways

Binary Number System

What would happen if only two digits would be used- 0 and 1? Or if we would not know to determine whether something is 3 or 5 times greater than something else? Or if we would be restricted when comparing two sizes, i.e. if we could only state that something exists (1) or does not exist (0)? Nothing special would happen, we would keep on using numbers in the same way, but they would look a bit different. For example: 11011010. How many pages of a book does the number 11011010 include? In order to learn that, follow the same logic like in the previous

example, but in reverse order. Bear in mind that all this is about mathematics with only two digits- 0 and 1, i.e. base-2 number system (binary number system).

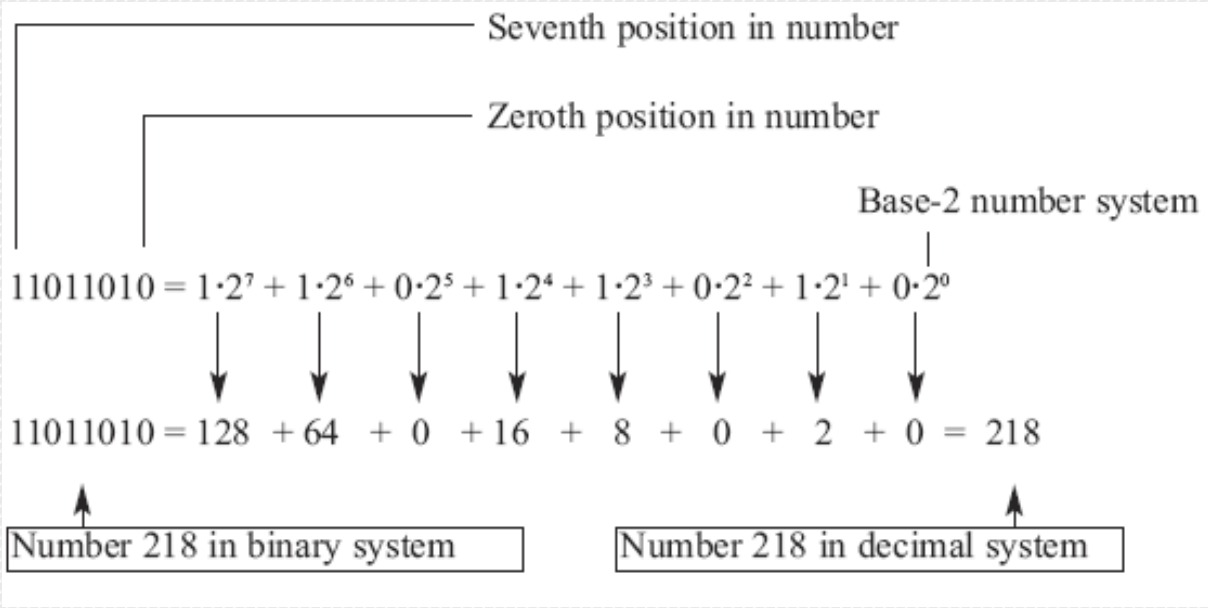


Fig. 0-3 The number 218 represented in binary and decimal system

Clearly, it is the same number represented in two different ways. The only difference is in the number of digits necessary for writing some number. One digit (2) is used to write the number 2 in decimal system, whereas two digits (1 and 0) are used to write that number in binary system. Do you now agree that there are 10 groups of people? Welcome to the world of binary arithmetic! Do you have any idea where it is used?

Excepting strictly controlled laboratory conditions, the most complicated electronic circuits cannot accurately determine the difference between two sizes (two voltage values, for example) if they are too small (lower than several volts). The reasons are electrical noises and something called the "real working environment" (unpredictable changes of power supply voltage, temperature changes, tolerance to values of built in components etc.). Imagine a computer which would operate upon decimal numbers by recognizing 10 digits in the following way: 0=0V, 1=5V, 2=10V, 3=15V, 4=20V... 9=45V !? Did anybody say batteries? A far simpler solution is the use of binary logic where 0 indicates that there is no voltage and 1 indicates that there is voltage. It is easier to write 0 or 1 instead of "there is no voltage" or "there is voltage". It is called logic zero (0) and logic one (1) which electronics perfectly conforms with and easily performs all those endlessly complex mathematical operations. It is electronics which in reality applies mathematics in which all numbers are represented by two digits only and in which it is only important to know whether there is voltage or not. Of course, we are talking about digital electronics.

Hexadecimal Number System

At the very beginning of computer development it was realized that people had many difficulties in handling binary numbers. Because of this, a new numbering system had to be established. This time, a number system using 16 different digits. The first ten digits are the same as digits we are used to (0, 1, 2, 3,... 9) but there are six digits more. In order to keep from making up new symbols, the six letters of alphabet A, B, C, D, E and F are used. A hexadecimal number system consisting of digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F has been established. What is the purpose of this seemingly bizarre combination? Just look how perfectly everything fits the story about binary numbers.

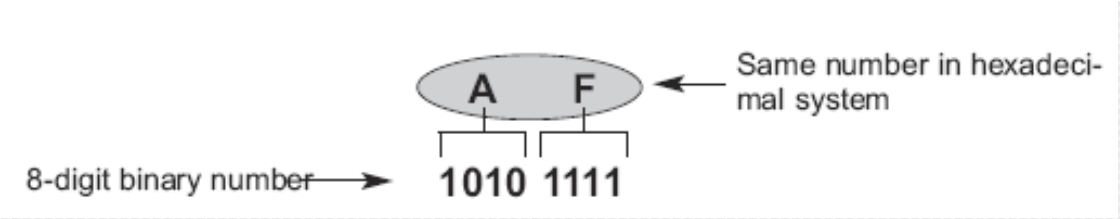


Fig. 0-4 Binary and Hexadecimal number

The largest number that can be represented by 4 binary digits is the number 1111. It corresponds to the number 15 in decimal system. That number is in hexadecimal system represented by only one digit F. It is the largest onedigit number in hexadecimal system. Do you see how skillfully it is used? The largest number written with eightdigits is at the same time the largest twodigit hexadecimal number. Bear in mind that the computer uses 8-digit binary numbers.

BCD Code

BCD code is actually a binary code for decimal numbers only. It is used to enable electronic circuits to communicate in a decimal number system with peripherals and in a binary system within “their own world”. It consists of fourdigit binary numbers which represent the first ten digits (0, 1, 2, 3 ... 8, 9). Even though four digits can give a total of 16 possible combinations, only the first ten are used.

Number System Conversion

The binary numbering system is the most commonly used, the decimal system is the most understandable while the hexadecimal system is somewhere between them. Therefore, it is very important to learn how to convert numbers from one numbering system to another, i.e. how to turn a series of zeros and units into values understandable to us.

Binary to Decimal Number Conversion

Digits in a binary number have different values depending on their position in that number. Additionally, each position can contain either 1 or 0 and its value may be easily determined by its position from the right. To make the conversion of a binary number to decimal it is necessary to multiply values with the corresponding digits (0 or 1) and add all the results. The magic of binary to decimal number conversion works...You doubt? Look at the example:

110 = 1*2^2 + 1*2^1 + 0*2^0 = 6

It should be noted that for decimal numbers from 0 to 3 you only need two binary digits. For greater values, extra binary digits must be added. Thus, for numbers from 0 to 7 you need three digits, for numbers from 0 to 15- four digits etc. Simply speaking, the largest binary number consisting of n digits is obtained when the base 2 is raised by n. The result should be then subtracted by 1. For example, if n=4:

2^4 - 1 = 16 - 1 = 15

Accordingly, using 4 binary digits it is possible to represent decimal numbers from 0 to 15, including these two digits, which amounts to 16 different values in total.

Hexadecimal to Decimal Number Conversion

In order to make conversion of a hexadecimal number to decimal, each hexadecimal digit should be multiplied with the number 16 raised by its position value. For example:

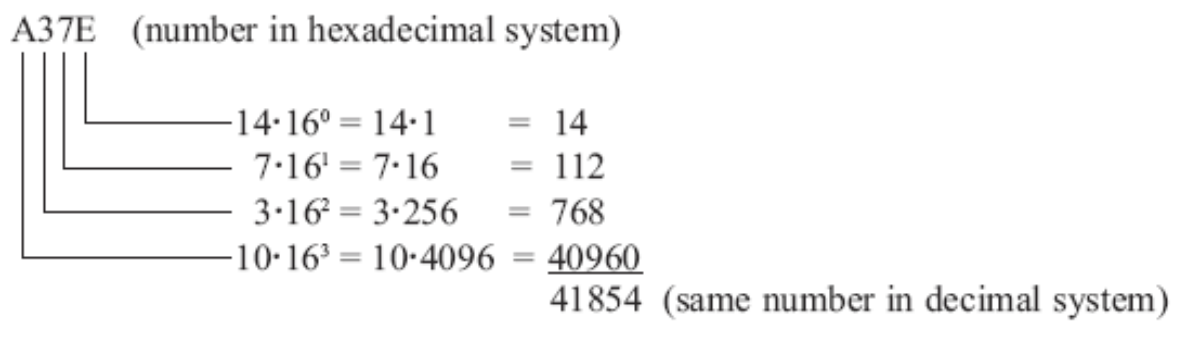


Fig. 0-5 Hexadecimal to decimal number conversion

Hexadecimal to Binary Number Conversion

It is not necessary to perform any calculation in order to convert hexadecimal numbers to binary numbers. Hexadecimal digits are simply replaced by the appropriate four binary digits. Since the maximum hexadecimal digit is equivalent to decimal number 15, we need to use four binary digits to represent one hexadecimal digit. For example:

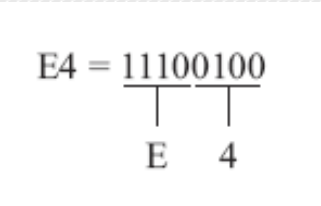


Fig. 0-6 Hexadecimal to binary number conversion

DEC.	BINARY								HEX.
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	3
4	0	0	0	0	0	1	0	0	4
5	0	0	0	0	0	1	0	1	5
6	0	0	0	0	0	1	1	0	6
7	0	0	0	0	0	1	1	1	7
8	0	0	0	0	1	0	0	0	8
9	0	0	0	0	1	0	0	1	9
10	0	0	0	0	1	0	1	0	A
11	0	0	0	0	1	0	1	1	B
12	0	0	0	0	1	1	0	0	C
13	0	0	0	0	1	1	0	1	D
14	0	0	0	0	1	1	1	0	E
15	0	0	0	0	1	1	1	1	F
16	0	0	0	1	0	0	0	0	10
17	0	0	0	1	0	0	0	1	11
	.....								
	.....								
	.....								
253	1	1	1	1	1	1	0	1	FD
254	1	1	1	1	1	1	1	0	FE
255	1	1	1	1	1	1	1	1	FF

Comparative table below contains the values of numbers 0-255 in three different numbering systems.

Marking Numbers

The hexadecimal numbering system is along with binary and decimal number systems considered to be the most important for us. It is easy to make conversion of any hexadecimal number to binary and it is also easy to remember it. However, these conversions may cause confusion. For example, what does the statement “It is necessary to count up 110 products on assembly line” actually mean? Depending on whether it is about binary, decimal or hexadecimal, the result could be 6, 110 or 272 products, respectively! Accordingly, in order to avoid misunderstanding, different prefixes and suffixes are directly added to the numbers. The prefix $ or 0x as well as the suffix h marks the numbers in hexadecimal system. For example, hexadecimal number 10AF may look as follows $10AF, 0x10AF or 10AFh. Similarly, binary numbers usually get the suffix % or 0b, whereas decimal numbers get the suffix D.

Bit

Theory says a bit is the basic unit of information... Let’s forget this dry explanation for a moment and take a look at what it is in practice. The answer is nothing special a bit is a binary digit. Similar to decimal number system in which digits in a number do not have the same value (for example digits in the number 444 are the same, but have different values), the “significance” of the bit depends on the position it has in the binary number. Therefore, there is no point talking about units, tens etc. Instead, here it is about the zero bit (rightmost bit), first bit (second from the right) etc. In addition, since the binary system uses two digits only (0 and 1), the value of one bit can be 0 or 1.

Don’t be confused if you find some bit has value 4, 16 or 64. It means that bit’s values are represented in decimal system. Simply, we have got so much accustomed to the usage of decimal numbers that these expressions became common. It would be correct to say for example, “the value of sixth bit in binary number is equivalent to decimal number 64”. But we are human and habits die hard... Besides, how would it sound “number: one-onezero- one-zero...”

Byte

A byte or a program word consists of eight bits grouped together. If a bit is a digit, it is logical that bytes represent numbers. All mathematical operations can be performed upon them, like upon common decimal numbers. As is the case with digits of any other number, byte digits do not have the same significance. The largest value has the leftmost bit called the most significant bit (MSB). The rightmost bit has the least value and

is therefore called the least significant bit (LSB). Since eight zeros and units of one byte can be combined in 256 different ways, the largest decimal number which can be represented by one byte is 255 (one combination represents zero).

A nibble is referred to as half a byte. Depending on which half of the byte we are talking about (left or right), there are “high” and “low” nibbles.

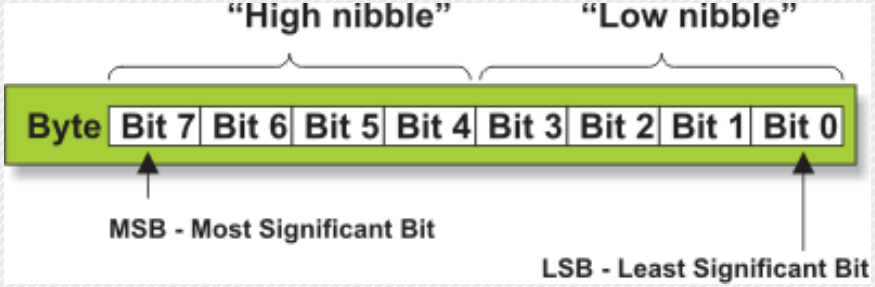


Fig. 0-8 High and Low nibbles

Logic Circuits

Have you ever wondered what electronics within some digital integrated circuits, microcontrollers or processors look like? What do the circuits performing complicated mathematical operations and making decisions look like? Do you know that their seemingly complicated schematics comprise only a few different elements called “logic circuits” or “logic gates”?

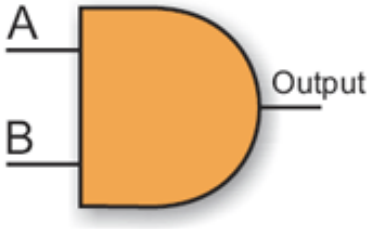
The operation of these elements is based on the principles established by British mathematician George Boole in the middle of the 19th century- even before the first bulb was invented! In brief, the main idea was to express logical forms through algebraic functions. Such thinking was soon transformed into a practical product which far later evaluated in what today is known as AND, OR and NOT logic circuits. The principle of their operation is known as Boolean algebra. As some program instructions used by the microcontroller perform the same way as logic gates except in the form of commands, the principle of their operation will be discussed here.

AND Gate

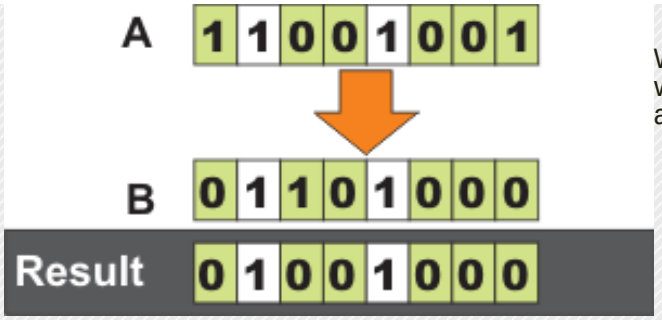
A logic gate “AND” has two or more inputs and one output. Let us presume that the gate used in this case has only two inputs. A logic one (1) will appear on its output only in case both inputs (A AND B) are driven to logic one (1).

The table shows mutual dependence between inputs and output.

When the gate has more than two inputs, the principle of operation is the same: a logic one (1) will appear on its output only if case all inputs are driven to logic one (1). Any other combination of input voltages will result in a logic zero (0) at its output.



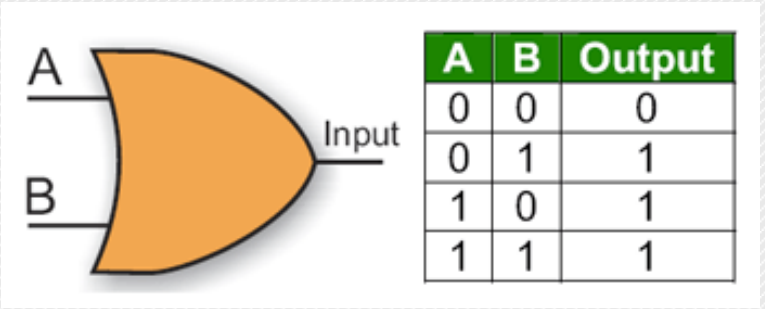
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1



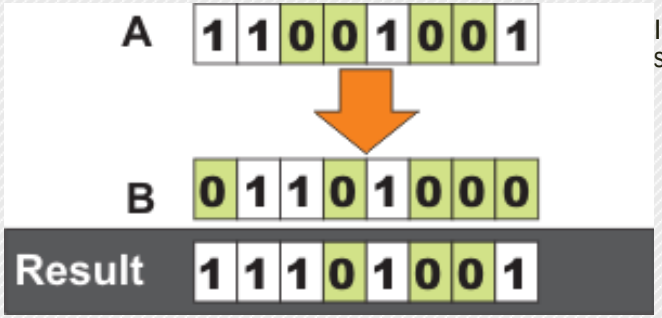
When used in a program, a logic AND operation is performed by the program instruction, which will be discussed later. For the time being, it is enough to remember that logic AND in a program refers to the corresponding bits of two registers.

OR Gate

Similarly to the previous case, OR gates also have two or more inputs and one output. A logic one (1) will appear on its output if either input (A OR B) is driven to logic one (1). If all inputs are at logic zero (0), the output will be driven to logic zero (0).



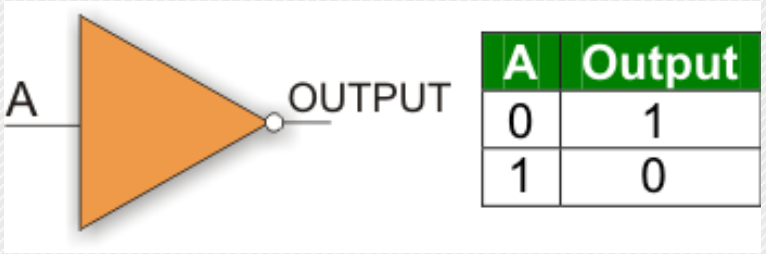
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1



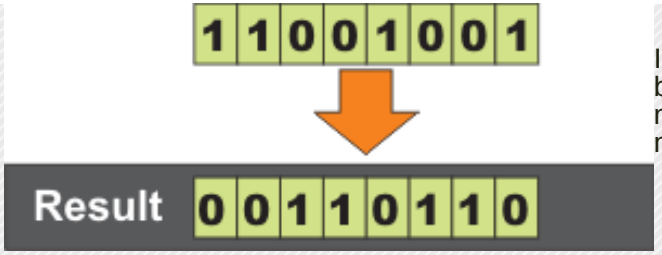
In a program, logic OR operation is performed between the corresponding registers' bits- the same as in logic AND operation.

NOT Gate

This logic gate has only one input and only one output. It operates in an extremely simple way. When logic zero (0) appears on its input, a logic one (1) appears on its output and vice versa. This means that this gate inverts the signal by itself. It is sometimes called inverter.



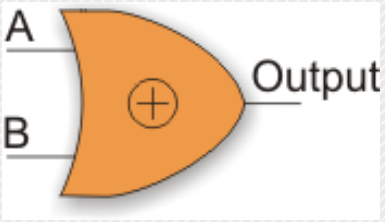
A	Output
0	1
1	0

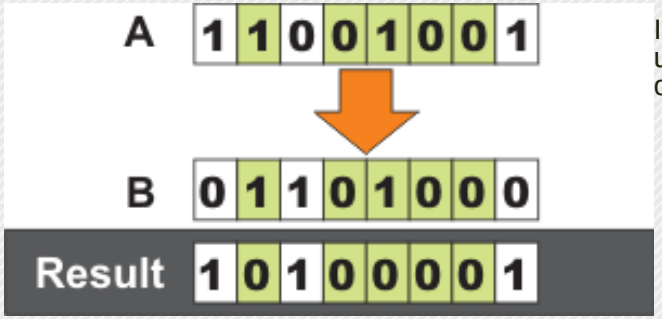


If a program, logic NOT operation is performed on one byte. The result is a byte with inverted bits. If byte is considered to be a number, the inverted value is actually a complement of that number, i.e. the complement of a number is what is needed to add to it to make it reach the maximal 8 bit value (255).

EXCLUSIVE OR Gate

The EXCLUSIVE OR (XOR) gate is a bit complicated comparing to other gates. It represents a combination of all the previously described gates. A logic one (1) appears on its output only when the inputs have different logic states.





In a program, this operation is commonly used to compare two bytes. Subtraction may be used for the same purpose (if the result is 0, bytes are equal). The advantage of this logic operation is that there is no danger to subtract larger number from smaller one.

Register

A register or a memory cell is an electronic circuit which can memorize the state of one byte.

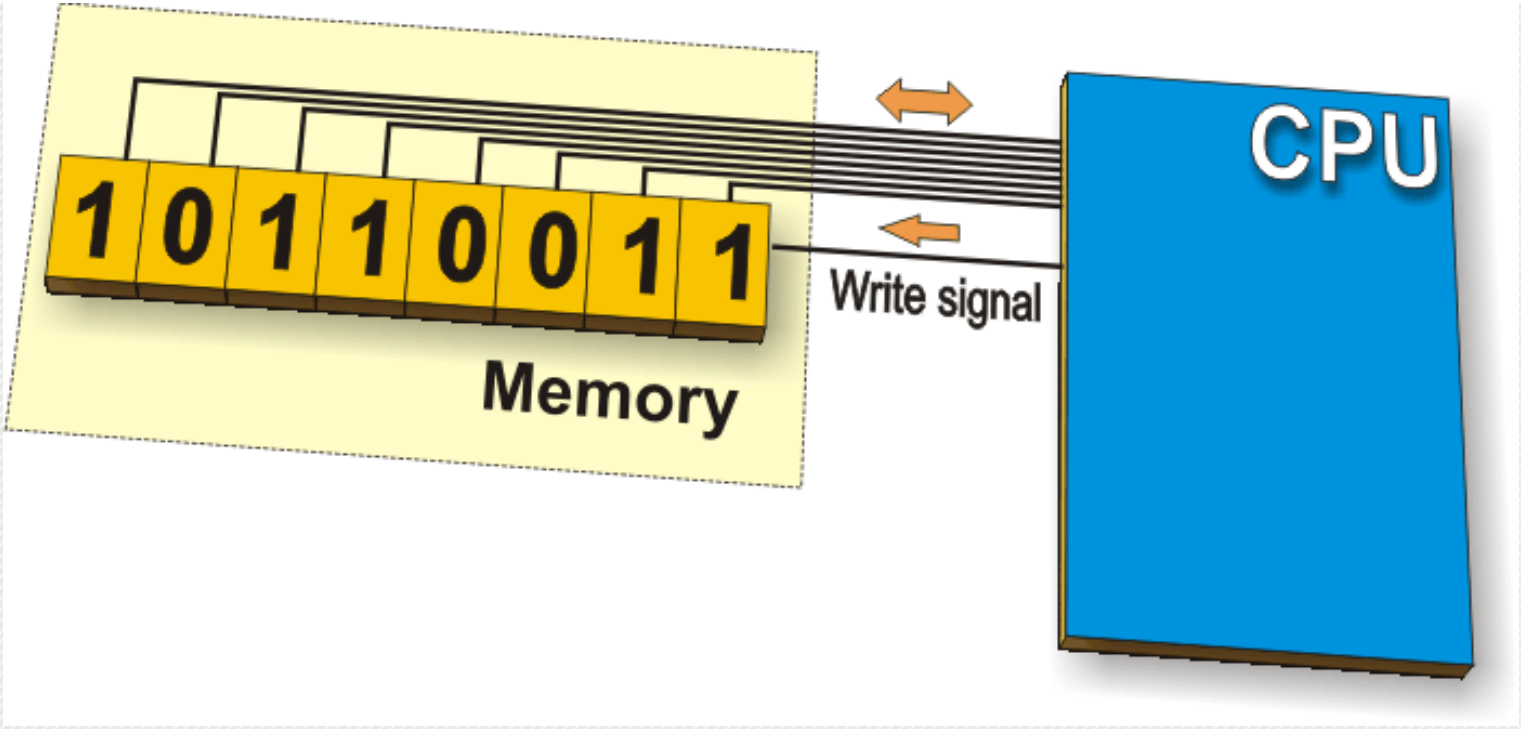


Fig. 0-17 Register

Special Function Register

In addition to the registers which do not have any special and predetermined function, every microcontroller has a number of registers whose function is predetermined by the manufacturer. Their bits are connected (literally) to internal circuits such as timers, A/D converter, oscillators and others, which means that they are directly in command of the operation of the microcontroller. Imagine eight switches which are in command of some smaller circuits within the microcontroller- you are right! Special Function Registers (SFRs) do exactly that!

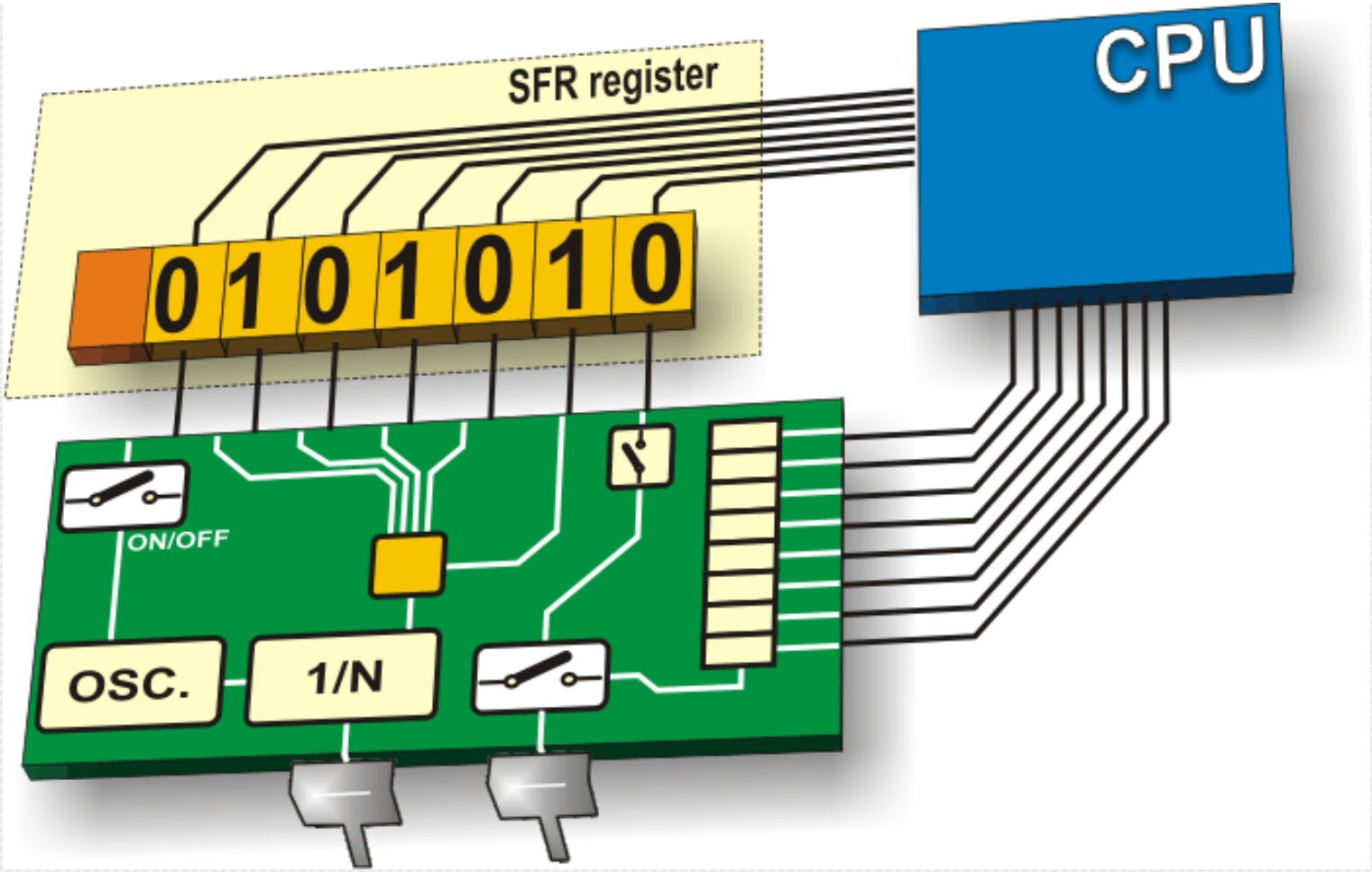


Fig. 0-18 Special Function Register

Input/Output Ports

In order to make the microcontroller useful, it has to be connected to additional electronics, i.e. peripherals. Each microcontroller has one or more registers (called a “port”) connected to the microcontroller pins. Why input/output? Because you can change the pin’s function as you wish. For example, suppose you want your device to turn three signal LEDs and simultaneously monitor the logic state of five sensors or push buttons. Some of ports need to be configured so that there are three outputs (connected to the LEDs) and five inputs (connected to sensors). It is simply performed by software, which means that the pin’s function can be changed during operation.

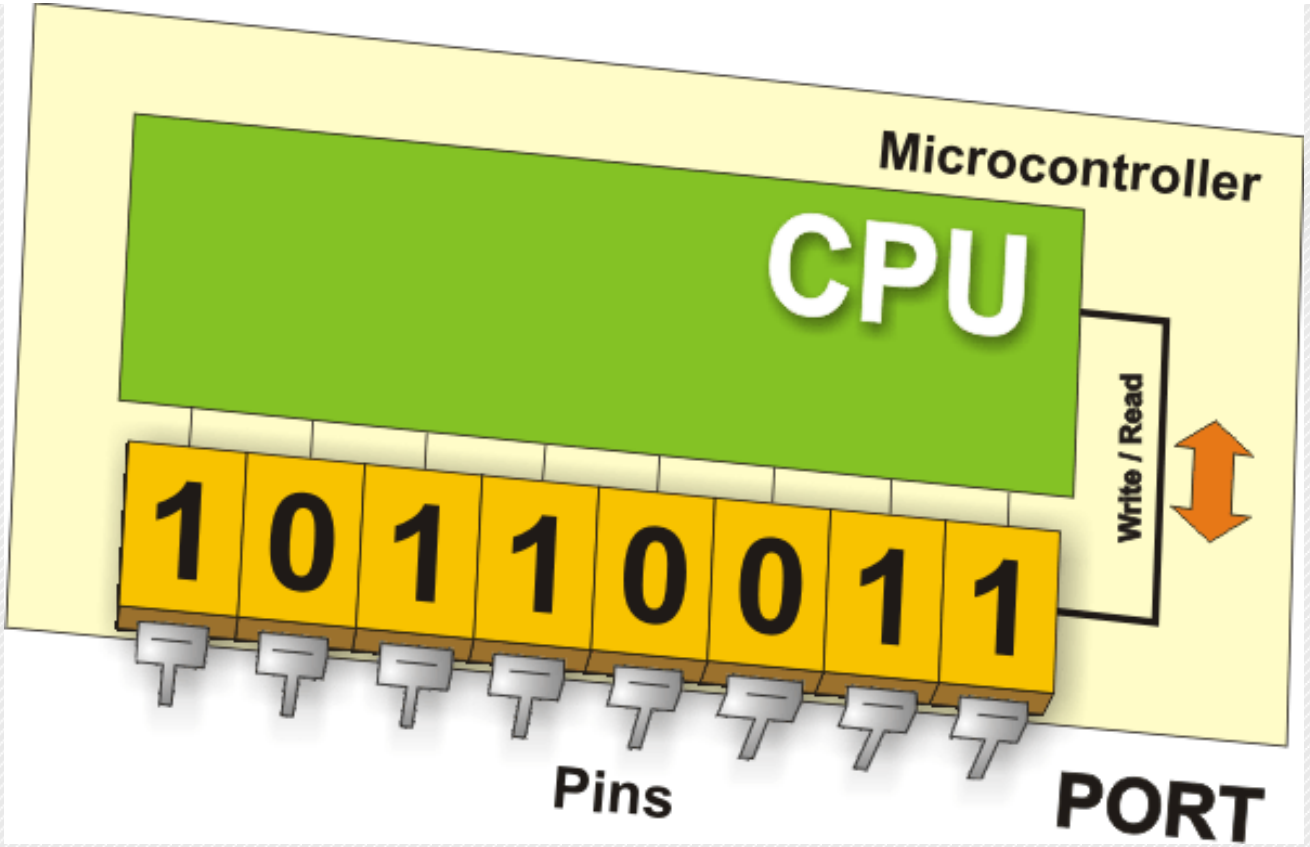


Fig. 0-19 Input / Output ports

One of the more important specifications of input/output (I/O) pins is the maximum current they can handle. For most microcontrollers, current obtained from one pin is sufficient to activate an LED or other similar low-current device (10-20 mA). If the microcontroller has many I/O pins, then the maximum current of one pin is lower. Simply put, you cannot expect all pins to give maximum current if there are more than 80 of them on one microcontroller. Another way of putting it is that the maximum current stated in the data specifications sheet for the microprocessor is shared across all I/O ports.

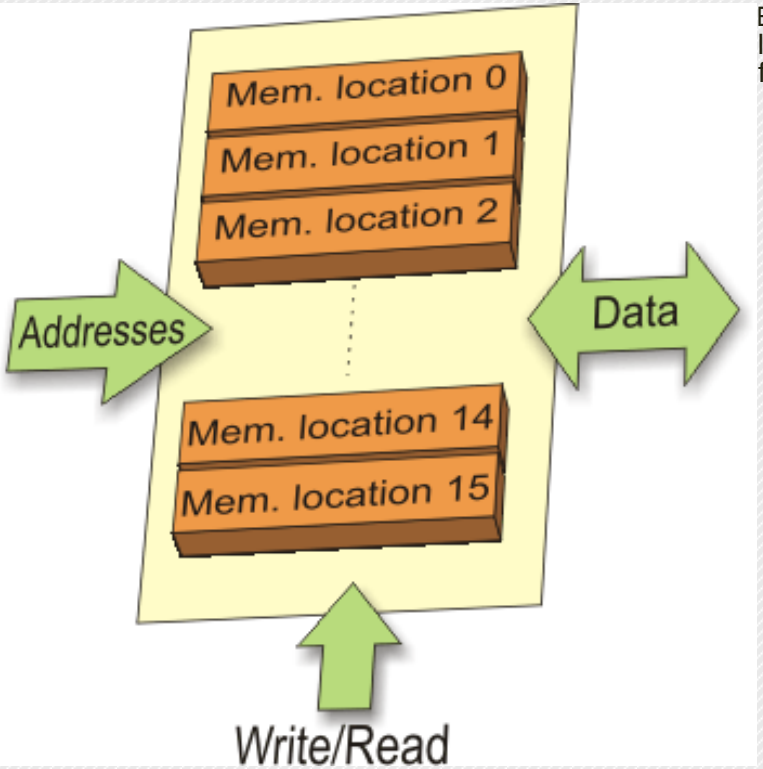
Another important pin function is that it can have pull-up resistors. These resistors connect pins to the positive power supply voltage and their effect is visible when the pin is configured as an input connected to mechanical switch or push button. Newer versions of microcontrollers have pull-up resistors configurable by software.

Usually, each I/O port is under control of another SFR, which means that each bit of that register determines the state of the corresponding microcontroller pin. For example, by writing logic one (1) to one bit of that control register SFR, the appropriate port pin is automatically configured as input. It means that voltage brought to that pin can be read as logic 0 or 1. Otherwise, by writing zero to the SFR, the appropriate port pin is configured as an output. Its voltage (0V or 5V) corresponds to the state of the appropriate bit of the port register.

Memory Unit

Memory is part of the microcontroller used for data storage. The easiest way to explain it is to compare it with a filing cabinet with many

drawers. Suppose, the drawers are clearly marked so that it is easy to access any of them. It is easy enough to find out the contents of the drawer by reading the label on the front of the drawer.



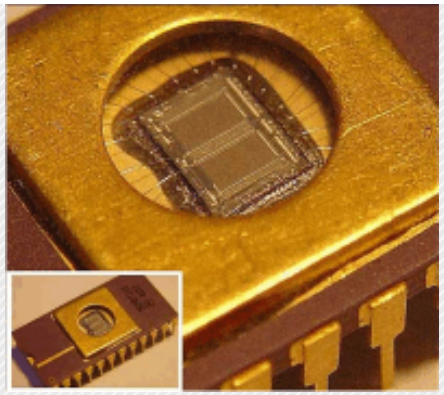
Each memory address corresponds to one memory location. The content of any location becomes known by its addressing. Memory can either be written to or read from. There are several types of memory within the microcontroller.

Read Only Memory (ROM)

ROM (Read Only Memory) is used to permanently save the program being executed. The size of a program that can be written depends on the size of this memory. Today's microcontrollers commonly use 16-bit addressing, which means that they are able to address up to 64 Kb of memory, i.e. 65535 locations. As a novice, your program will rarely exceed the limit of several hundred instructions. There are several types of ROM.

Masked ROM. Microcontrollers containing this ROM are reserved for the great manufacturers. Program is loaded into the chip by the manufacturer. In case of large scale manufacture, the price is very low. Forget it...

One Time Programmable ROM (OTP ROM). If the microcontroller contains this memory, you can download a program into this memory, but the process of program downloading is a "one-way ticket", meaning that it can be done only once. If an error is detected after downloading, the only thing you can do is to download the corrected program to another chip.



UV Erasable Programmable ROM (UV EPROM). Both the manufacturing process and characteristics of this memory are completely identical to OTP ROM. However, the package of this microcontroller has a recognizable “window” on the upper side. It enables the surface of the silicon chip inside to be lit by an UV lamp, which effectively erases and program from the ROM.
Installation of this window is very complicated, which normally affects the price. From our point of view, unfortunately- negative...

Flash memory. This type of memory was invented in the 80s in the laboratories of INTEL and were represented as the successor to the UV EPROM. Since the contents of this memory can be written and cleared practically an unlimited number of times, the microcontrollers with Flash ROM are ideal for learning, experimentation and small-scale manufacture. Because of its popularity, the most microcontrollers are manufactured in flash versions today. So, if you are going to buy a microcontroller, the type to look for is definitely Flash!

Random Access Memory (RAM)
Once the power supply is off the contents of RAM (Random Access Memory) is cleared. It is used for temporary storing data and intermediate results created and used during the operation of the microcontroller. For example, if the program performs an addition (of whatever), it is necessary to have a register representing what in everyday life is called the “sum”. For that purpose, one of the registers in RAM is called the “sum” and used for storing results of addition.

Electrically Erasable Programmable ROM (EEPROM)
The contents of the EEPROM may be changed during operation (similar to RAM), but remains permanently saved even upon the power supply goes off (similar to ROM). Accordingly, an EEPROM is often used to store values, created during operation, which must be permanently saved. For example, if you design an electronic lock or an alarm, it would be great to enable the user to create and enter a password, but useless if it is lost every time the power supply goes off. The ideal solution is the microcontroller with an embedded EEPROM.

Interrupt

The most programs use interrupts in regular program execution. The purpose of the microcontroller is mainly to react on changes in its surrounding. In other words, when some event takes place, the microcontroller does something... For example, when you push a button on a remote controller, the microcontroller will register it and respond to the order by changing a channel, turn the volume up or down etc. If the microcontroller spent most of its time endlessly a few buttons for hours or days... It would not be practical.

The microcontroller has learnt during its evolution a trick. Instead of checking each pin or bit constantly, the microcontroller delegates the “wait issue” to the “specialist” which will react only when something attention worthy happens.

The signal which informs the central processor about such an event is called an INTERRUPT.

Central Processor Unit (CPU)

As its name suggests, this is a unit which monitors and controls all processes inside the microcontroller. It consists of several smaller subunits, of which the most important are:

- **Instruction Decoder** is a part of the electronics which recognizes program instructions and runs other circuits on the basis of that. The "instruction set" which is different for each microcontroller family expresses the abilities of this circuit.
- **Arithmetical Logical Unit (ALU)** performs all mathematical and logical operations upon data.
- **Accumulator** is a SFR closely related to the operation of the ALU. It is a kind of working desk used for storing all data upon which some operation should be performed (addition, shift/move etc.). It also stores the results ready for use in further processing. One of the SFRs, called a Status Register (PSW), is closely related to the accumulator. It shows at any given moment the "status" of a number stored in the accumulator (number is greater or less than zero etc.).

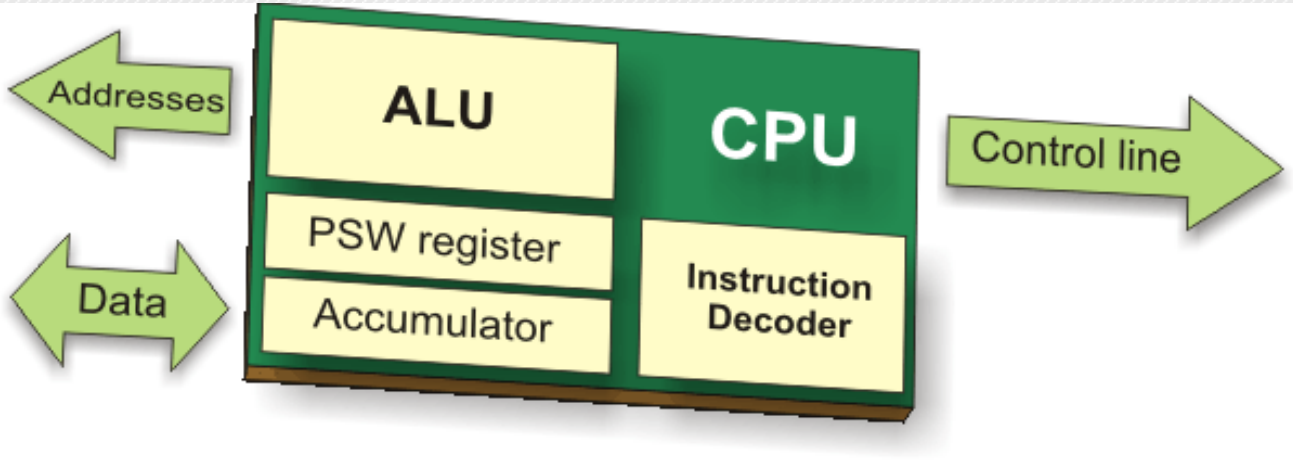


Fig. 0-22 Central Processor Unit - CPU

Bus

Physically, the bus consists of 8, 16 or more wires. There are two types of buses: the address bus and the data bus. The address bus consists of as many lines as necessary for memory addressing. It is used to transmit the address from the CPU to the memory. The data bus is as wide as the data, in our case it is 8 bits or wires wide. It is used to connect all circuits inside the microcontroller.

Serial Communication

Parallel connections between the microcontroller and peripherals via input/output ports is the ideal solution for shorter distances- up to several meters. However, in other cases - when it is necessary to establish communication between two devices on longer distances it is not possible to use a parallel connection - such a simple solution is out of question. In these situations, serial communication is the best solution.

Today, most microcontrollers have built in several different systems for serial communication as a standard equipment. Which of these systems will be used depends on many factors of which the most important are:

- How many devices the microcontroller has to exchange data with?
- How fast the data exchange has to be?
- What is the distance between devices?
- Is it necessary to send and receive data simultaneously?

One of the most important things concerning serial communication is the *Protocol* which

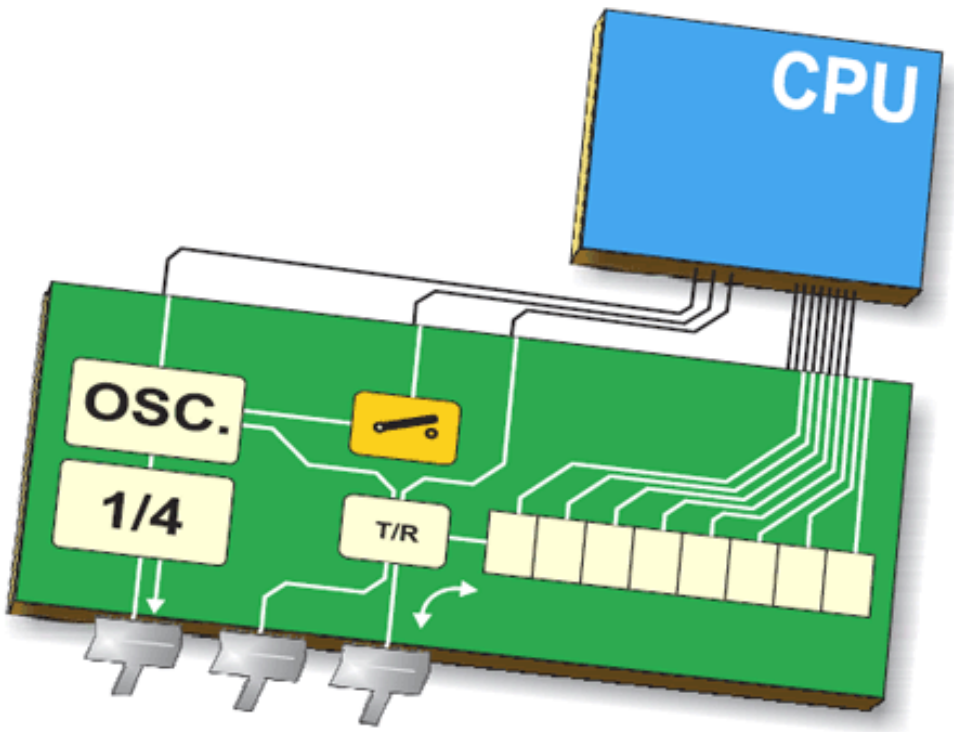


Fig. 0-23 Serial communication

should be strictly observed. It is a set of rules which must be applied in order that the devices can correctly interpret data they mutually exchange. Fortunately, the microcontrollers automatically take care of this, so the work of the programmer/user is reduced to simple write (data to be sent) and read (received data).

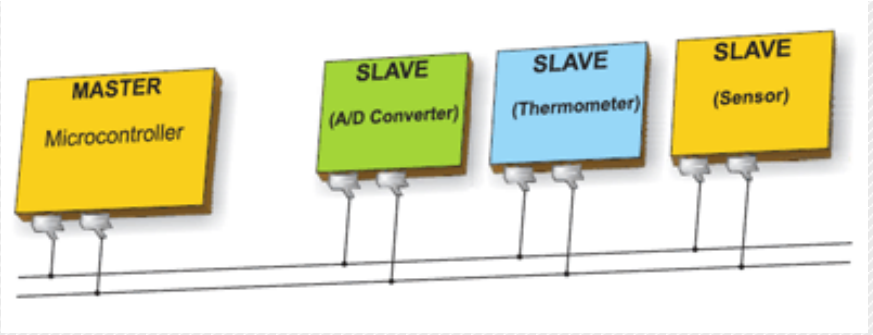
Baud Rate

The term *Baud rate* is commonly used to denote the number of bits transferred per second [bps].

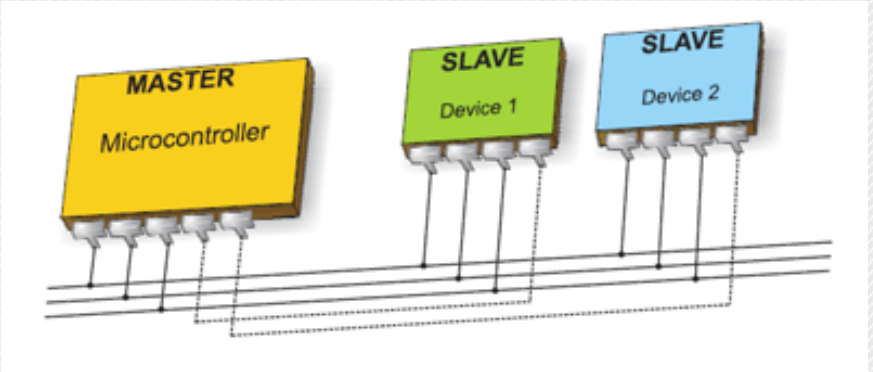
It should be noted that it refers to bits, not bytes! It is usually required by the protocol that each byte is transferred along with several control bits. It means that one byte in serial data stream may consist of 11 bits. For example, if the baud rate is 300 bps then maximum 37 and minimum 27 bytes may be transferred per second, which depends on type of connection and protocol in use.

The most commonly used serial communication systems are:

I2C (Inter Integrated Circuit) is a system used when the distance between the microcontrollers is short and specialized integrated circuits of of a new generation (receiver and transmitter are usually on the same printed circuit board). Connection is established via two conductors- one is used for data transfer whereas another is used for synchronization (clock signal). As seen in figure, one device is always the master. It performs addressing of one slave chip (subordinated) before communication starts. In this way one microcontroller can communicate with 112 different devices. Baud rate is usually 100 Kb/sec (standard mode) or 10 Kb/sec (slow baud rate mode). Systems with the baud rate of 3.4 Mb/sec have recently appeared. The distance between devices which communicate via an inter-integrated circuit bus is limited to several meters.



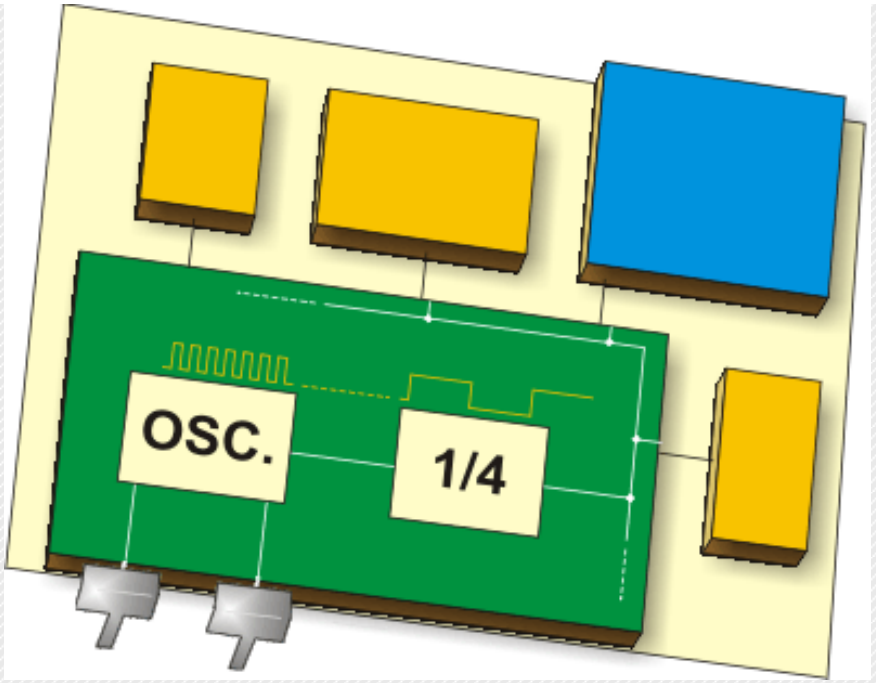
SPI (Serial Peripheral Interface Bus) is a system for serial communication which uses up to four conductors (usually three)- one for data receiving, one for data sending, one for synchronization and one (alternatively) for selecting the device to communicate with. It is full duplex connection, which means that data is sent and received simultaneously. The maximum baud rate is higher than in I2C connection.



UART (Universal Asynchronous Receiver/Transmitter)
This connection is asynchronous, which means that a special line for clock signal transmission is not used. In some situations this feature is crucial (for example, radio connection or infrared waves remote control). Since only one communication line is used, both receiver and transmitter operate at the same predefined rate in order to maintain necessary synchronization. This is a very simple way of transferring data since it basically represents conversion of 8-bit data from parallel to serial format. Baud rate is not high up to 1 Mbit/sec.

Oscillator

Even pulses coming from the oscillator enable harmonic and synchronous operation of all circuits of the microcontroller. The oscillator module is usually configured to use quartz crystal or ceramic resonator for frequency stabilization. Furthermore, it can also operate without elements for frequency stabilization (like RC oscillator). It is important to say that instructions are not executed at the rate imposed by the oscillator itself, but several times slower. It happens because each instruction is executed in several steps. In some microcontrollers, the same number of cycles is needed to execute any instruction, while in others, the execution time is not the same for all instructions. Accordingly, if the system uses quartz crystal with a frequency of 20 Mhz, execution time of an instruction is not 50nS, but 200, 400 or 800 nS, depending on the type of Microcontroller Unit (MCU)!



Power supply circuit

There are two things worth attention concerning the microcontroller power supply circuit:

Brown-out is a potentially dangerous state which occurs at the moment the microcontroller is being turned off or in situations when power supply voltage drops to the limit due to electric noise. As the microcontroller consists of several circuits which have different operating voltage levels, this state can cause its out-of-control performance. In order to prevent it, the microcontroller usually has built-in circuit for brown out reset. This circuit immediately resets the whole electronics when the voltage level drops below the limit.

Reset pin is usually marked as MCLR (Master Clear Reset) and serves for external reset of the microcontroller by applying logic zero (0) or one (1), depending on type of the microcontroller. In case the brown out circuit is not built in, a simple external circuit for brown out reset can be connected to this pin.

Timers/Counters

The microcontroller oscillator uses quartz crystal for its operation. Even though it is not the simplest solution, there are many reasons to use it. Namely, the frequency of such oscillator is precisely defined and very stable, the pulses it generates are always of the

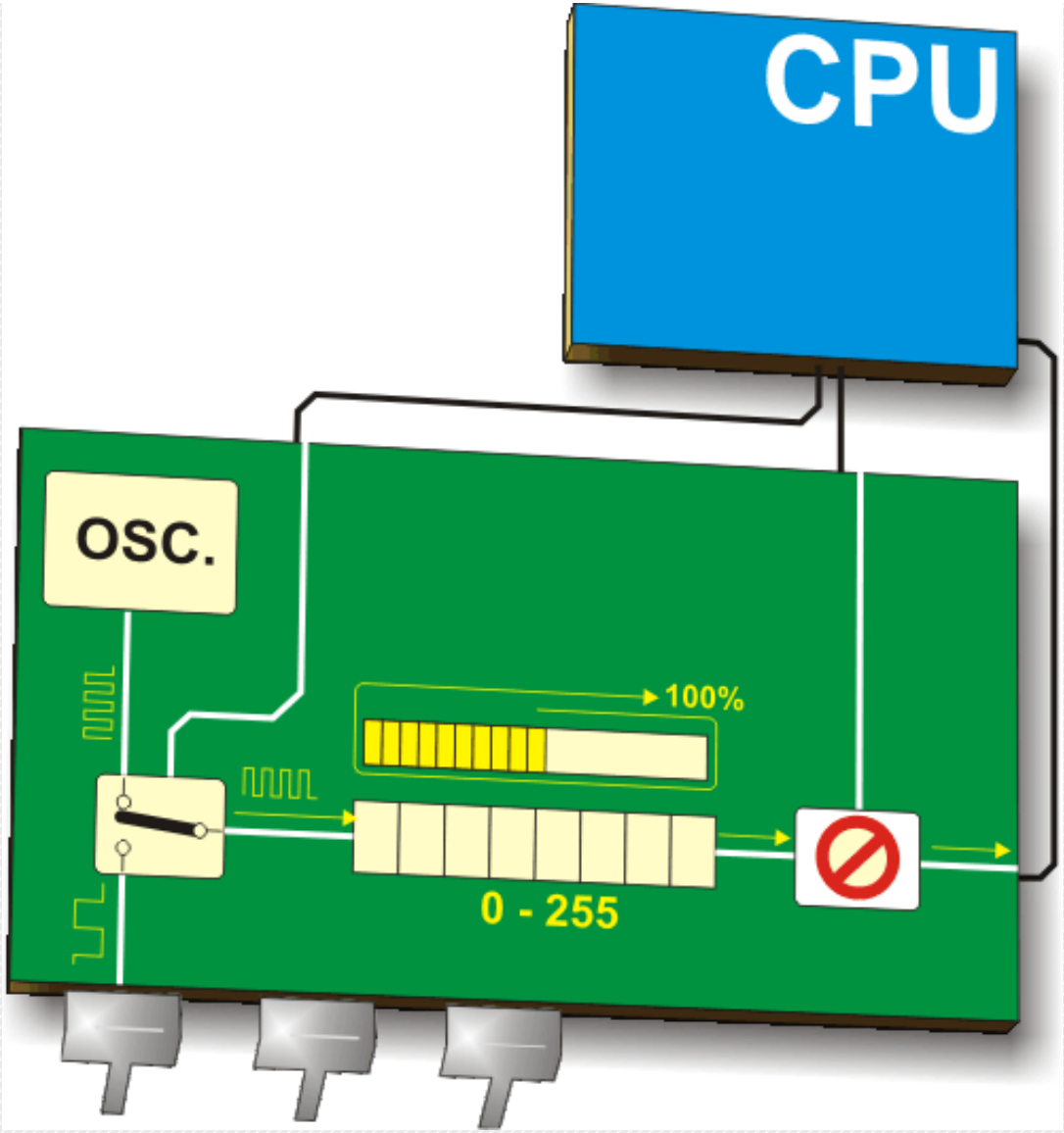


Fig. 0-27 Timers/Counters

same width, which makes them ideal for time measurement. Such oscillators are used in quartz watches. If it is necessary to measure time between two events, it is sufficient to count pulses coming from this oscillator. That is exactly what the timer does.

Most programs use these miniature electronic “stopwatches”. These are commonly 8- or 16-bit SFRs and their content is automatically incremented by each coming pulse. Once a register is completely loaded - an interrupt is generated!

If the timer registers use an internal quartz oscillator for their operation then it is possible to measure time between two events (if the register value is T1 at the moment measurement has started, and T2 at the moment it has finished, then the elapsed time is equal to the result of

subtraction T2-T1). If the registers use pulses coming from external source then such a timer is turned into a counter.

This is only a simple explanation of the operation itself.

How does a timer operate?

In practice pulses coming from the quartz oscillator are once per each machine cycle directly or via a prescaler brought to the circuit which increments the number in the timer register. If one instruction (one machine cycle) lasts for four quartz oscillator periods then, by embedding quartz with the frequency of 4MHz, this number will be changed a million times per second (each microsecond).

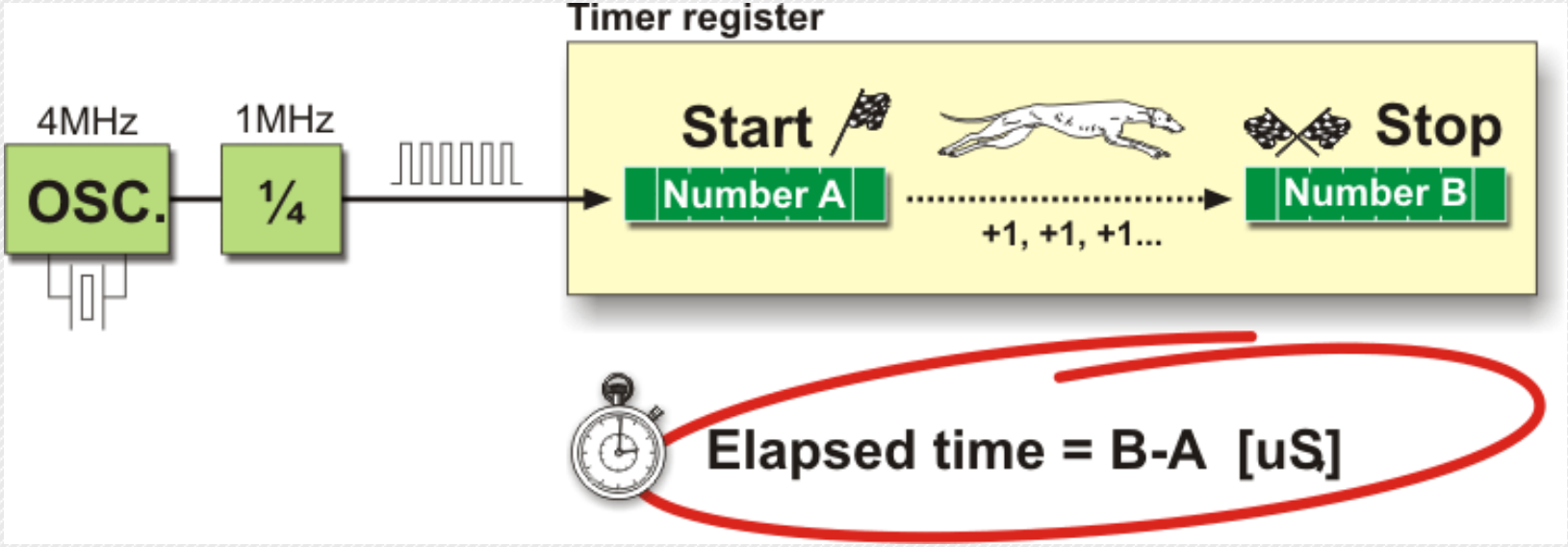


Fig. 0-28 Timer Operation

It is easy to measure short time intervals (up to 256 microseconds) in the way described above because it is the largest number that one register can contain. This obvious disadvantage may be easily overcome in several ways by using a slower oscillator, registers with more bits, a prescaler or interrupts. The first two solutions have some weaknesses so it is preferable to use prescalers or interupts.

Using prescaler in timer operating

A prescaler is an electronic device used to reduce a frequency by a pre-determined factor. Meaning that in order to generate one pulse on its output, it is necessary to bring 1, 2 , 4 or more pulses to its input. One such circuit is built in the microcontroller and its division rate can be changed from within the program. It is used when it is necessary to measure longer periods of time.

One prescaler is usually shared by timer and watch-dog timer, which means that it cannot be used by both of them simultaneously.

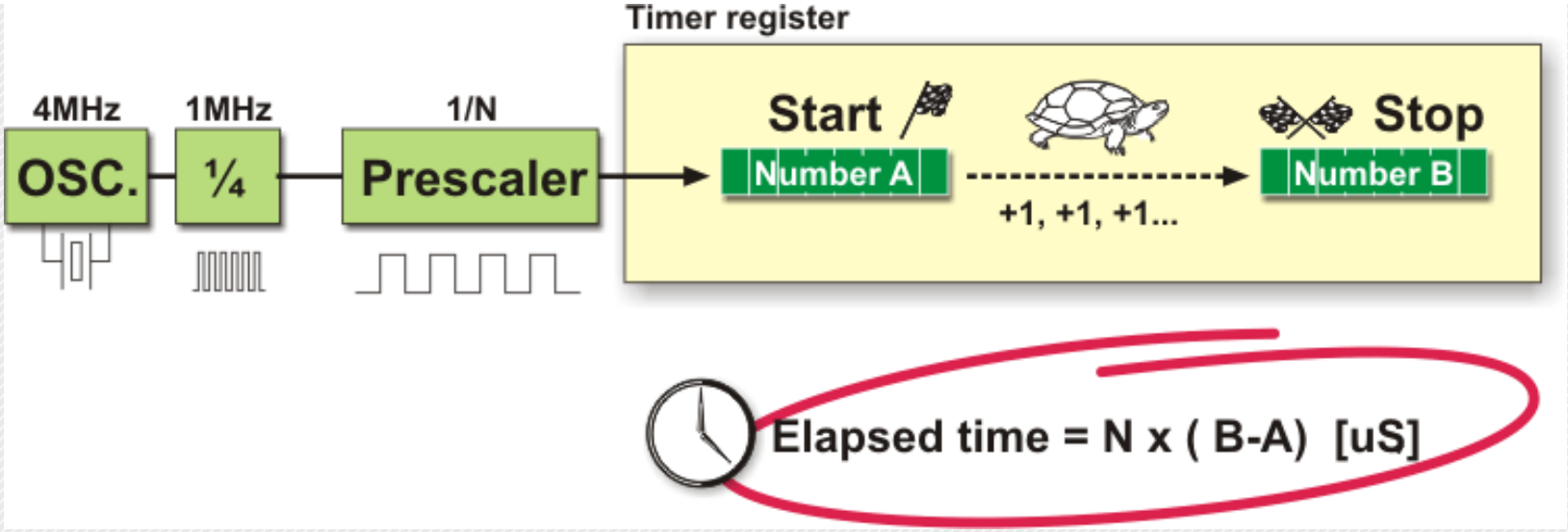


Fig. 0-29 Using prescaler in timer operating

Using the interrupt in timer operation

If the timer register consists of 8 bits, the largest number that can be written to it is 255 (for 16-bit registers it is the number 65.535). If this number is exceeded, the timer will be automatically reset and counting will start from zero again. This condition is called overflow. If enabled from within the program, such overflow can cause an interrupt, which gives completely new possibilities. For example, the state of registers used for counting seconds, minutes or days can be changed in an interrupt routine. The whole process (except interrupt routine) is automatically performed “in the background”, which enables the main circuits of the microcontroller to perform other operations.

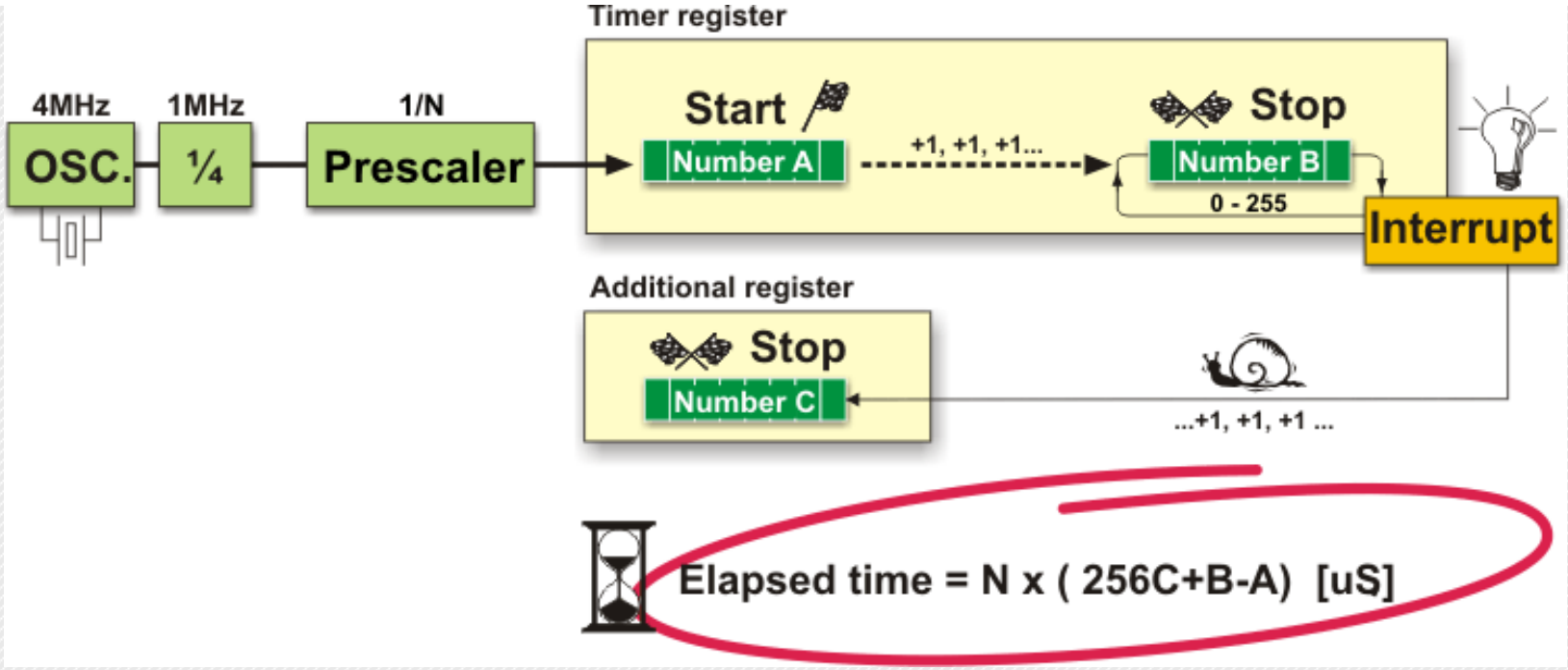


Fig. 0-30 Using the interrupt in timer operation

This figure illustrates the use of the interrupt in timer operation. Delays of arbitrary duration with minimal interference by the main program execution can be easily obtained by assigning a prescaler to the timer.

Counters

If a timer is supplying pulses into the microcontroller input pin then it turns into a counter. Clearly, It is the same electronic circuit. The only difference is that in this case pulses to be counted come through the ports and their duration (width) is mostly not defined. This is why they cannot be used for time measurement, but can be used to measure anything else: products on an assembly line, number of axis rotation, passengers etc. (depending on sensor in use).

Watchdog Timer

The Watchdog Timer is a timer connected to a completely separate RC oscillator within the microcontroller.

If the watchdog timer is enabled, every time it counts up to the program end, the microcontroller reset occurs and program execution starts from the first instruction. The point is to prevent this from happening by using a specific command. The whole idea is based on the fact that every program is executed in several longer or shorter loops.

If instructions which reset the watchdog timer are set at the appropriate program locations, besides commands being regularly executed, then the operation of the watchdog timer will not affect program execution. If for any reason (usually electrical noises in industry), the program counter “gets stuck” on some memory location from which there is no return, the watchdog will not be cleared and the register’s value being constantly incremented will reach the maximum et voila! Reset occurs!

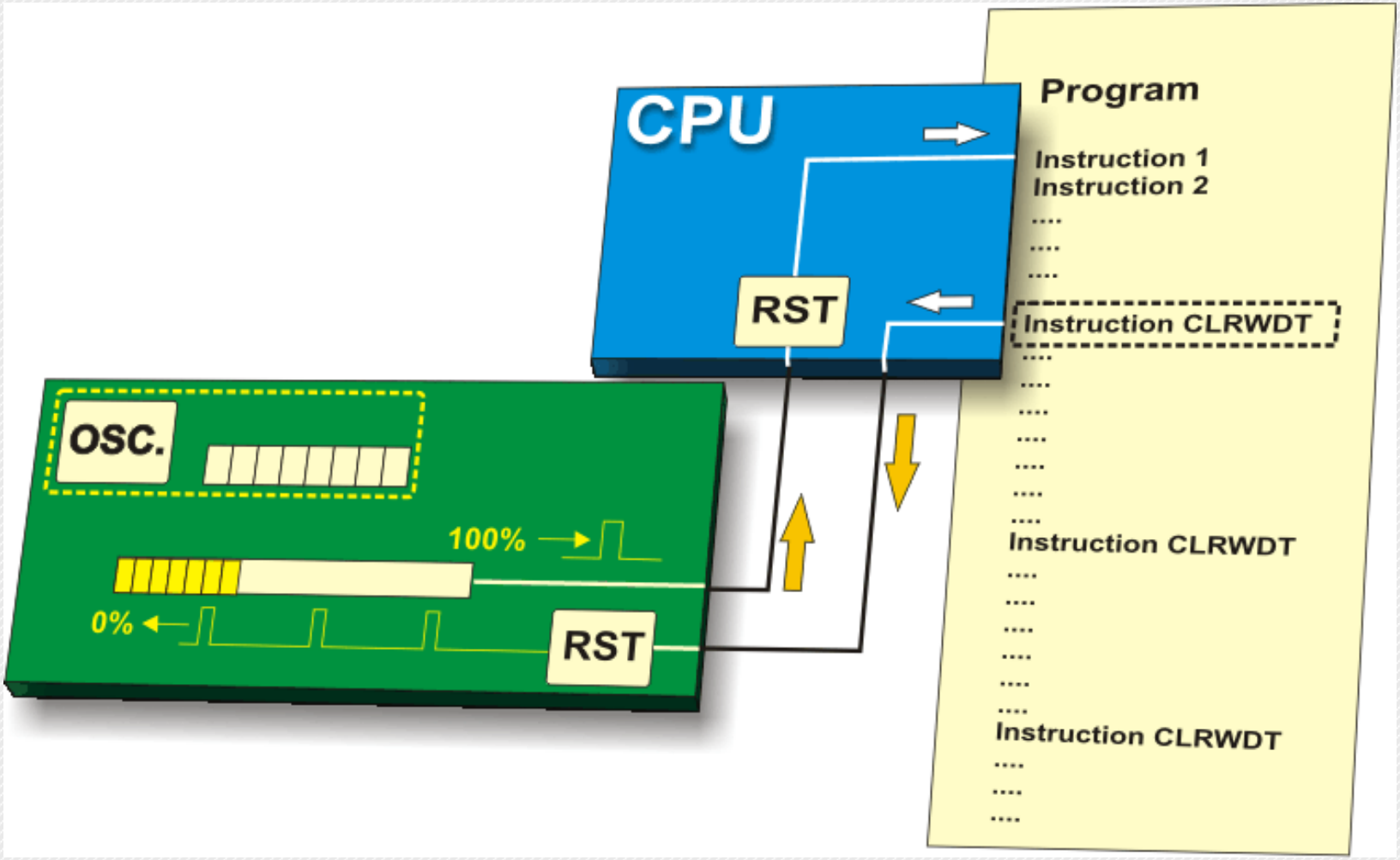


Fig. 0-31 Watchdog Timer

A/D Converter

External signals are usually fundamentally different from those the microcontroller understands (Ones and Zeros), so that they have to be converted in order for the microcontroller to understand them. An analogue to digital converter is an electronic circuit which converts continuous signals to discrete digital numbers. This module is therefore used to convert some analogue value into binary number and forwards it to the CPU for further processing. In other words, this module is used for input pin voltage measurement (analogue value). The result of measurement is a number (digital value) used and processed later in the program.

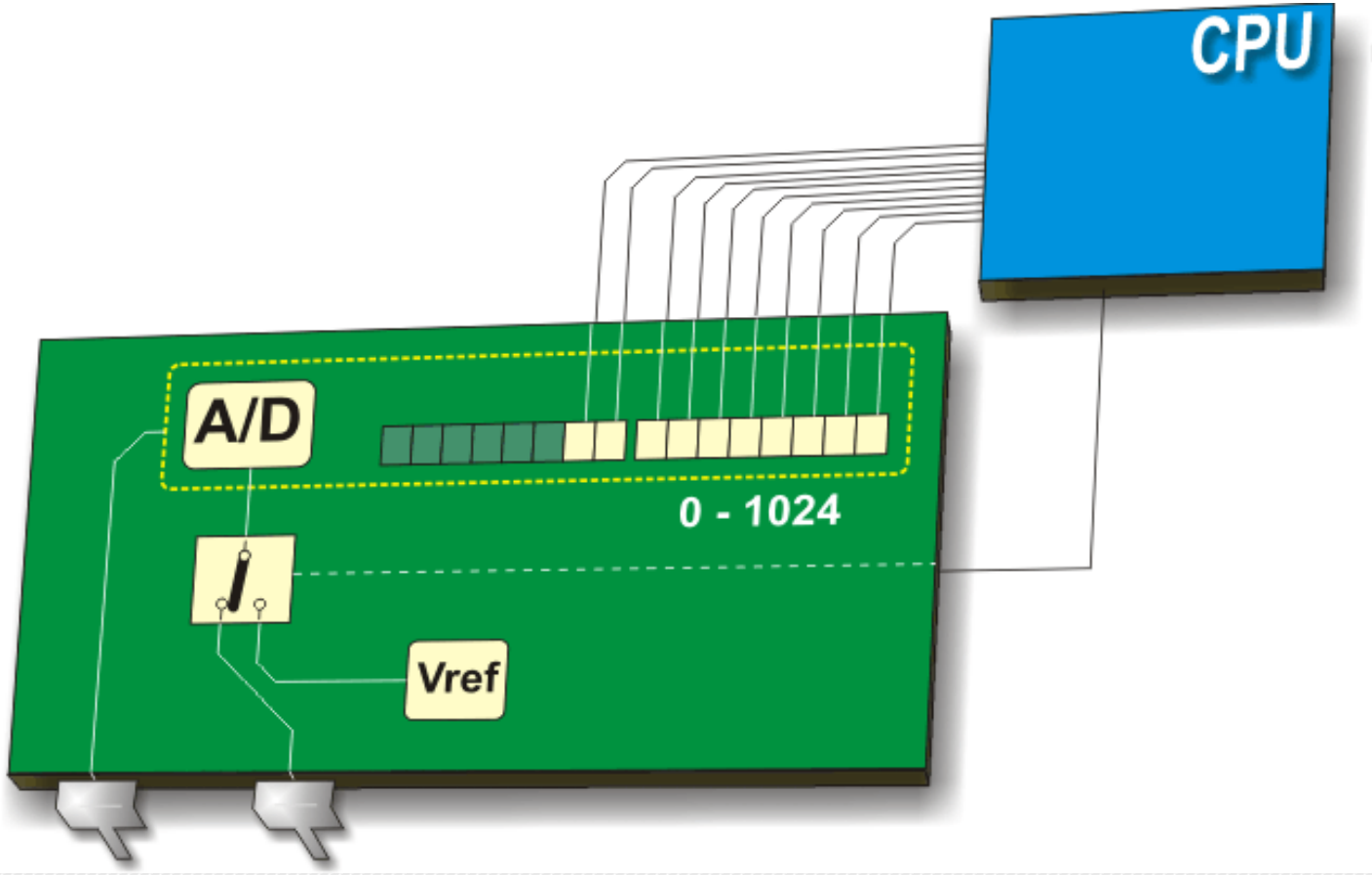


Fig. 0-32 A/D Converter

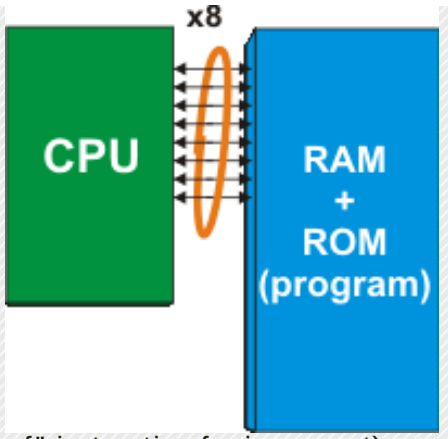
Internal Architecture

All upgraded microcontrollers use one of two basic design models called Harvard and von-Neumann architecture.

Briefly, they are two different ways of data exchange between CPU and memory.

von-Neumann Architecture

Microcontrollers using this architecture have only one memory block and one 8-bit data bus. As all data are exchanged by using these 8 lines, this bus is overloaded and communication itself is very slow and inefficient. The CPU can either read an instruction or read/write data from/to the memory. Both cannot occur at the same time since the instructions and data use the same bus system. For example, if some program line says that RAM memory register called "SUM" should be incremented by one (instruction: incf SUM), the microcontroller will do the following:

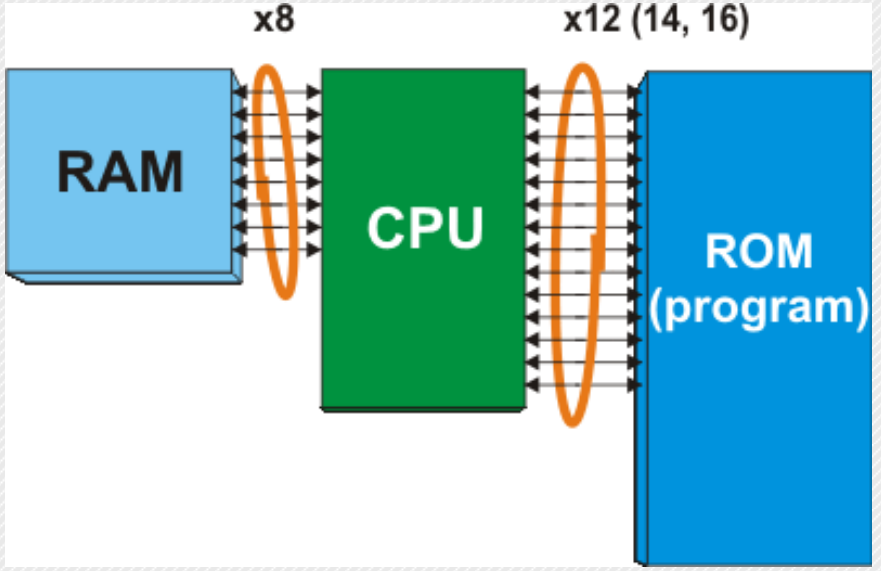


1. Read the part of the program instruction specifying WHAT should be done (in this very case it is the "incf" instruction for increment).
2. Read further the same instruction specifying upon WHICH data it should be performed (in this very case it is the "SUM" register).
3. After being incremented, the contents of this register should be written to the register from which it was read ("SUM" register address).

The same data bus is used for all these intermediate operations.

Harvard Architecture

Microcontrollers using this architecture have two different data buses. One is 8 bits wide and connects CPU to RAM. Another consists of several lines (12, 14 or 16) and connects CPU to ROM. Accordingly, the CPU can read an instruction and perform a data memory access at the same time. Since all RAM memory registers are 8 bits wide, all data within the microcontroller are exchanged in the same such format. Additionally, during program writing, only 8 bits data are considered. In other words, all you can ever change from within the program and all you can affect will be 8 bits wide. A program written for some of these microcontrollers will be stored in the microcontroller internal ROM upon having being compiled into machine language. However, these memory locations do not have 8, but 12, 14 or 16 bits. The rest of bits- 4, 6 or 8- represents the instruction itself specifying to the CPU what to do with the 8-bit data.

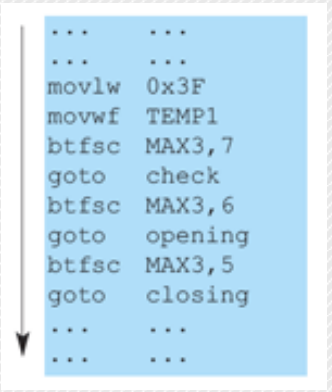


The advantages of such design are the following:

- All data in a program is one byte (8 bit) wide. As the data bus used for program reading has several lines (12, 14 or 16), both instructions and data can be read simultaneously by using these spare bits. Therefore, all instructions are executed in only one instruction cycle. The only exception is jump instruction which is executed in two cycles.
- Owing to the fact that a program (ROM) and temporary data (RAM) are separate, the CPU can execute two instructions simultaneously. Simply, while RAM read or write is in progress (the end of one instruction), the next program instruction is being read via another bus.
- When using microcontrollers with von-Neumann architecture one never knows how much memory is to be occupied by some program. Basically, each program instruction occupies two memory locations (one contains information on WHAT should be done, whereas another

contains information upon WHICH data it should be done). However, it is not a hard and fast rule, but the most common case. In microcontrollers with Harvard architecture, the program bus is wider than one byte, which allows each program word to consist of instruction and data. In other words: one program word- one instruction.

INSTRUCTION SET



Instructions that can be understood by the microcontroller are known as an instruction set. When you write a program in assembly language, you actually “tell a story” by specifying instructions in the order they should be executed. The main restriction in this process is the number of available instructions. The manufacturers stick to one of the two following strategies:

RISC *(Reduced Instruction Set Computer)*

In this case, the microcontroller recognizes and executes only basic operations (addition, subtraction, copying etc.). All other more complicated operations are performed by combining these (for example, multiplication is performed by performing successive addition). The constrains are obvious (try by using only a few words, to explain to someone how to reach the airport in some other city). However, there are also some great advantages. First of all, this language is easy to learn. Besides, the microcontroller is very fast so that it is not possible to see all the arithmetic “acrobatics” it performs. The user can only see the final result of all those operations. At last, it is not so difficult to explain where the airport is if you use the right words. For example: left, right, kilometers etc.

CISC *(Complex Instruction Set Computer)*

CISC is the opposite of RISC! Microcontrollers designed to recognize more than 200 different instructions can do much and are very fast. However, one needs to understand how to take all that such a rich language offers, which is not at all easy...

How to make the right choice

Ok, you are the beginner and you have made a decision to go on an adventure of working with the microcontrollers. Congratulations on your choice! However, it is not as easy to choose the right microcontroller as it may seem. The problem is not a limited range of devices, but the opposite!

Before you start designing some device based on the microcontroller, think of the following: how many input/output lines will I need for operation? Should it perform some other operations than to simply turn relays on/off? Does it need some specialized module such as serial communication, A/D converter etc. When you create a clear picture of what you need, the selection range is considerably reduced, then it is time to think of price. Is your plan to have several same devices? Several hundred? A million? Anyway, you get the point...

If you think of all these things for the very first time then everything seems a bit confusing. For that reason, go step by step. First of all, select the manufacturer, i.e. the family of the microcontrollers you can easily obtain. After that, study one particular model. Learn as much as you need, do not go into details. Solve a specific problem and something incredible will happen- you will be able to handle any model belonging to that family.

Remember learning to ride a bicycle: after several unavoidable bruises at the beginning, you will manage to keep balance and will be able to easily ride any other bicycle. And of course, you will never forget the skill in programming just as you will never forget riding bicycles!

PIC microcontrollers

PIC microcontrollers designed by Microchip Technology are likely the right choice for you if you are the beginner. Here is why...

The real name of this microcontroller is PICmicro (Peripheral Interface Controller), but it is better known as PIC. Its first ancestor was designed in 1975 by General Instruments. This chip called PIC1650 was meant for totally different purposes. About ten years later, by adding EEPROM memory, this circuit was transformed into a real PIC microcontroller. Nowadays, Microchip Technology announces a manufacturing of the 5 billionth sample...

In order that you can better understand the reasons for its popularity, we will briefly describe several important things.

Family	ROM [Kbytes]	RAM [bytes]	Pins	Clock Freq. [MHz]	A/D Inputs	Resolution of A/D Converter	Compar-ators	8/16 - bit Timers	Serial Comm.	PWM Outputs	Others
Base-Line 8 - bit architecture, 12-bit Instruction Word Length											
PIC10FXXX	0.375 - 0.75	16 - 24	6 - 8	4 - 8	0 - 2	8	0 - 1	1 x 8	-	-	-
PIC12FXXX	0.75 - 1.5	25 - 38	8	4 - 8	0 - 3	8	0 - 1	1 x 8	-	-	EEPROM
PIC16FXXX	0.75 - 3	25 - 134	14 - 44	20	0 - 3	8	0 - 2	1 x 8	-	-	EEPROM
PIC16HVXXX	1.5	25	18 - 20	20	-	-	-	1 x 8	-	-	Vdd = 15V
Mid-Range 8 - bit architecture, 14-bit Instruction World Length											
PIC12FXXX	1.75 - 3.5	64 - 128	8	20	0 - 4	10	1	1 - 2 x 8 1 x 16	-	0 - 1	EEPROM
PIC12HVXXX	1.75	64	8	20	0 - 4	10	1	1 - 2 x 8 1 x 16	-	0 - 1	-
PIC16FXXX	1.75 - 14	64 - 368	14 - 64	20	0 - 13	8 or 10	0 - 2	1 - 2 x 8 1 x 16	USART I2C SPI	0 - 3	-
PIC16HVXXX	1.75 - 3.5	64 - 128	14 - 20	20	0 - 12	10	2	2 x 8 1 x 16	USART I2C SPI	-	-
High-End 8 - bit architecture, 16-bit Instruction Word Length											
PIC18FXXX	4 - 128	256 - 3936	18 - 80	32 - 48	4 - 16	10 or 12	0 - 3	0 - 2 x 8 2 - 3 x 16	USB2.0 CAN2.0 USART I2C SPI	0 - 5	-
PIC18FXXJXX	8 - 128	1024 - 3936	28 - 100	40 - 48	10 - 16	10	2	0 - 2 x 8 2 - 3 x 16	USB2.0 USART Ethernet I2C SPI	2 - 5	-
PIC18FXXKXX	8 - 64	768 - 3936	28 - 44	64	10 - 13	10	2	1 x 8 3 x 16	USART I2C SPI	2	-

All PIC microcontrollers use harvard architecture, which means that their program memory is connected to CPU via more than 8 lines. Depending on the bus width, there are 12-, 14- and 16-bit microcontrollers. The table above shows the main features of these three categories.

As seen in the table on the previous page, excepting “16-bit monsters”- PIC 24FXXX and PIC 24HXXX- all PIC microcontrollers have 8-bit harvard architecture and belong to one out of three large groups. Therefore, depending on the size of a program word there are first, second and third category, i.e. 12-, 14- or 16-bit microcontrollers. Having similar 8- bit core, all of them use the same instruction set and the basic hardware ‘skeleton’ connected to more or less peripheral units.

In order to avoid tedious explanations and endless story about the useful features of different microcontrollers, this book describes the operation of one particular model belonging to “high middle class”. It is about PIC16F887- powerful enough to be worth attention and simple enough to be easily presented to everybody.

Table of Contents | Next Chapter

Chapter 1: PIC16F887 Microcontroller - Device Overview

The PIC16F887 is one of the latest products from *Microchip*. It features all the components which modern microcontrollers normally have. For its low price, wide range of application, high quality and easy availability, it is an ideal solution in applications such as: the control of different processes in industry, machine control devices, measurement of different values etc. Some of its main features are listed below.

- **RISC architecture**
  - Only 35 instructions to learn
  - All single-cycle instructions except branches

- **Operating frequency 0-20 MHz**
- **Precision internal oscillator**
  - Factory calibrated
  - Software selectable frequency range of 8MHz to 31KHz

- **Power supply voltage 2.0-5.5V**
  - Consumption: 220uA (2.0V, 4MHz), 11uA (2.0 V, 32 KHz) 50nA (stand-by mode)

- **Power-Saving Sleep Mode**
- **Brown-out Reset (BOR) with software control option**
- **35 input/output pins**
  - High current source/sink for direct LED drive
  - software and individually programmable *pull-up* resistor
  - Interrupt-on-Change pin

- **8K ROM memory in FLASH technology**
  - Chip can be reprogrammed up to 100.000 times

- ***In-Circuit Serial Programming* Option**
  - Chip can be programmed even embedded in the target device

- **256 bytes EEPROM memory**
  - Data can be written more than 1.000.000 times

- **368 bytes RAM memory**
- **A/D converter:**
  - 14-channels

- 10-bit resolution

- **3 independent timers/counters**
- **Watch-dog timer**
- **Analogue comparator module with**
  - Two analogue comparators
  - Fixed voltage reference (0.6V)
  - Programmable on-chip voltage reference

- **PWM output steering control**
- **Enhanced USART module**
  - Supports RS-485, RS-232 and LIN2.0
  - Auto-Baud Detect

- **Master Synchronous Serial Port (MSSP)**
  - supports SPI and I2C mode

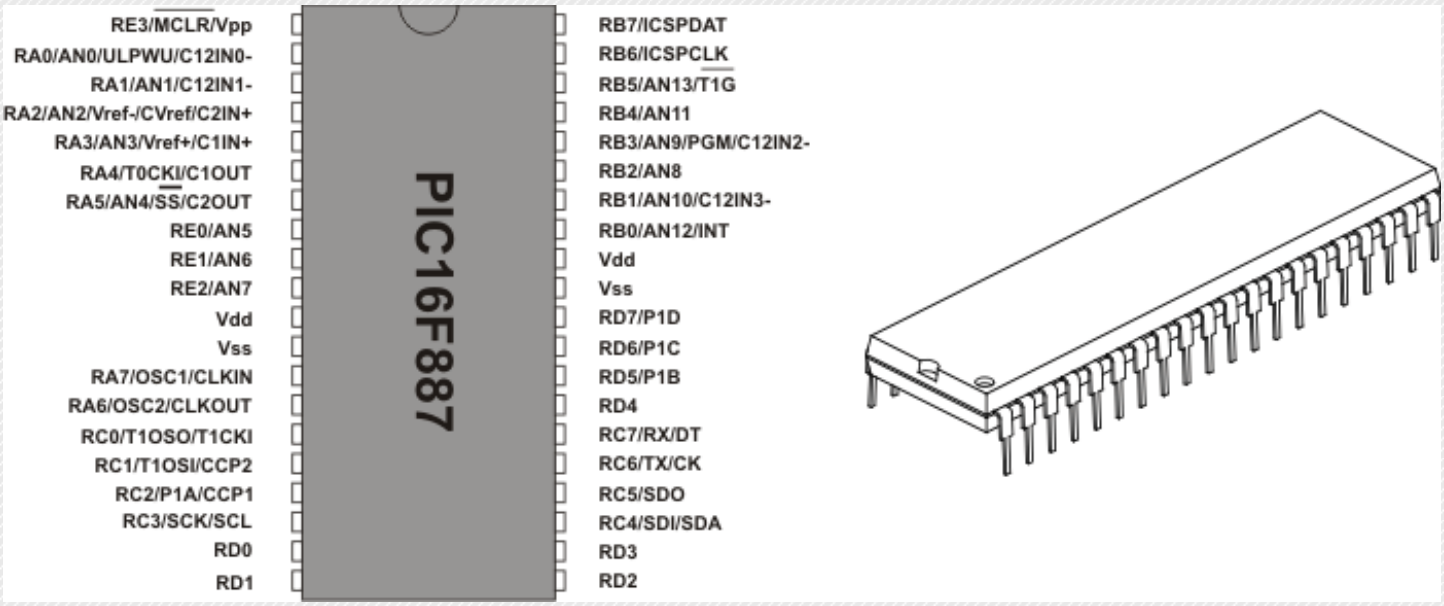


Fig. 1-1 PIC16F887 PDIP 40 Microcontroller

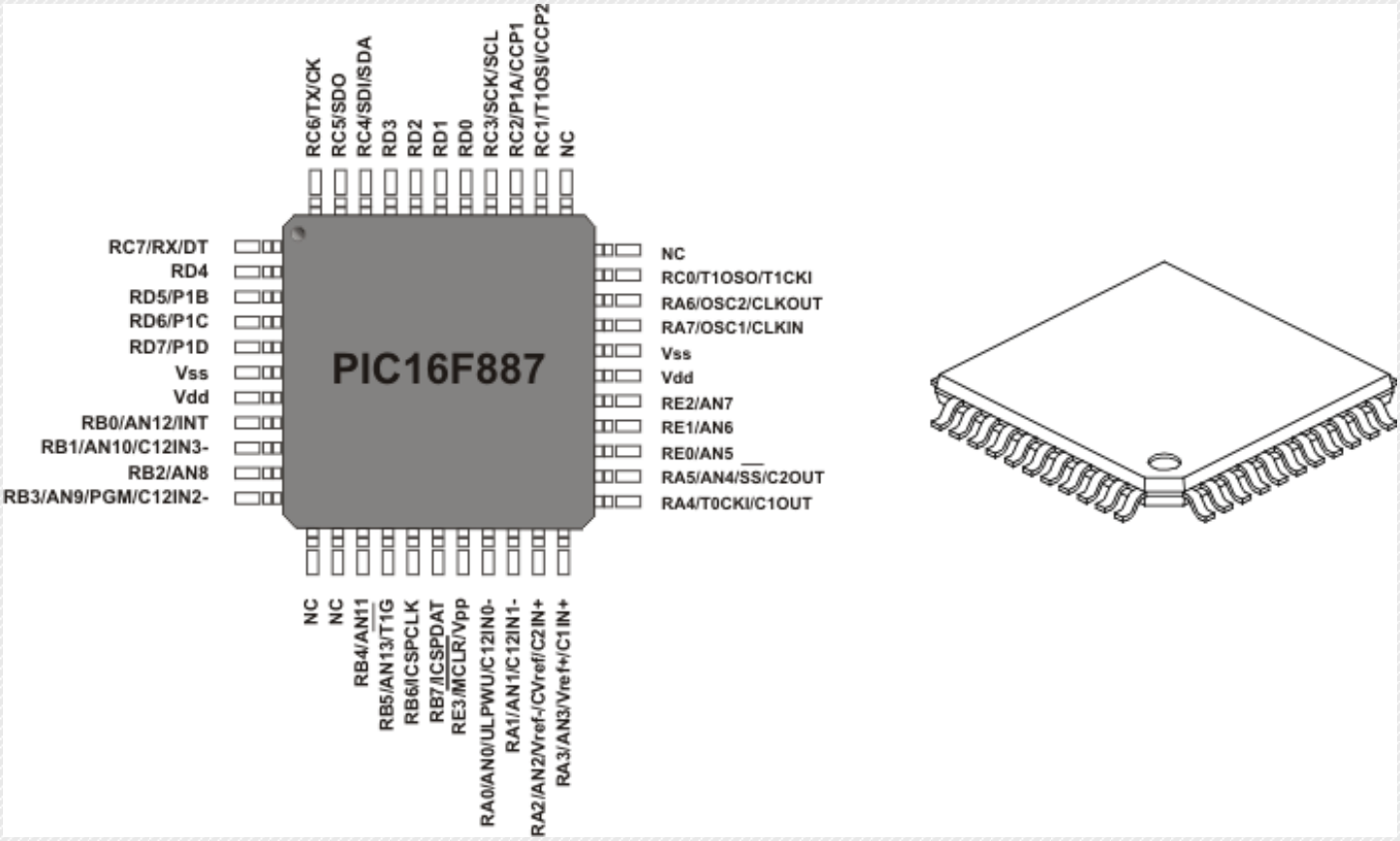


Fig. 1-2 PIC16F887 QFN 44 Microcontroller

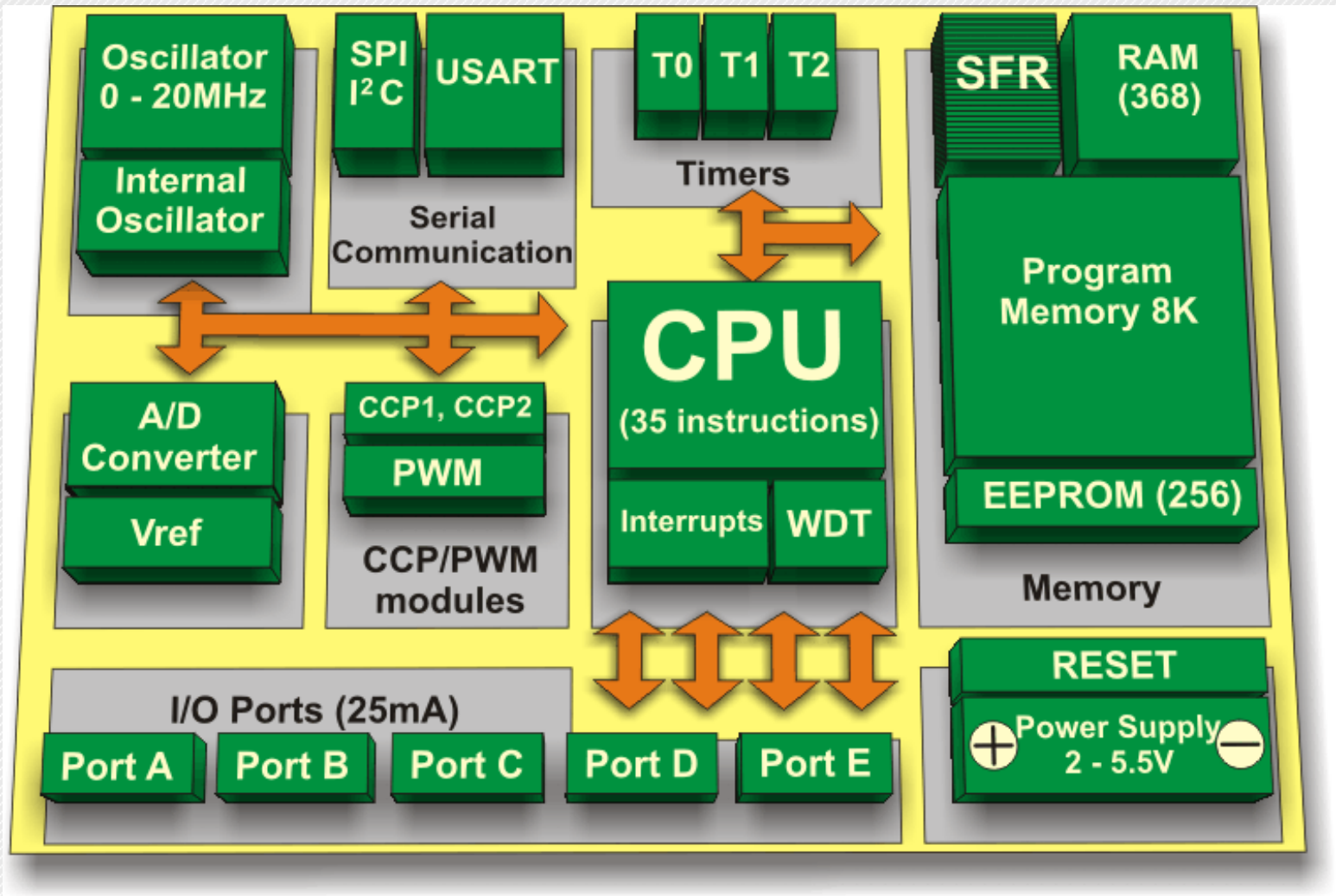


Fig. 1-3 PIC16F887 Block Diagram

Pin Description

As seen in Fig. 1-1 above, the most pins are multi-functional. For example, designator RA3/AN3/Vref+/C1IN+ for the fifth pin specifies the following functions:

- RA3 Port A third digital input/output
- AN3 Third analog input
- Vref+ Positive voltage reference
- C1IN+ Comparator C1positive input

This small trick is often used because it makes the microcontroller package more compact without affecting its functionality. These various pin functions cannot be used simultaneously, but can be changed at any point during operation.

The following tables, refer to the PDIP 40 microcontroller.

Name	Number (DIP 40)	Function	Description
RE3/MCLR/Vpp	1	RE3	General purpose input Port E
		MCLR	Reset pin. Low logic level on this pin resets microcontroller.
		Vpp	Programming voltage
RA0/AN0/ULPWU/C12IN0-	2	RA0	General purpose I/O port A
		AN0	A/D Channel 0 input
		ULPWU	Stand-by mode deactivation input
		C12IN0-	Comparator C1 or C2 negative input
RA1/AN1/C12IN1-	3	RA1	General purpose I/O port A
		AN1	A/D Channel 1
		C12IN1-	Comparator C1 or C2 negative input
RA2/AN2/Vref-/CVref/C2IN+	4	RA2	General purpose I/O port A
		AN2	A/D Channel 2
		Vref-	A/D Negative Voltage Reference input
		CVref	Comparator Voltage Reference Output
		C2IN+	Comparator C2 Positive Input
RA3/AN3/Vref+/C1IN+	5	RA3	General purpose I/O port A
		AN3	A/D Channel 3
		Vref+	A/D Positive Voltage Reference Input
		C1IN+	Comparator C1 Positive Input
RA4/T0CKI/C1OUT	6	RA4	General purpose I/O port A
		T0CKI	Timer T0 Clock Input
		C1OUT	Comparator C1 Output
		RA5	General purpose I/O port A
		AN4	A/D Channel 4

RA5/AN4/SS/C2OUT	7	RA5	General purpose I/O port A
		AN4	A/D Channel 4
		SS	SPI module Input (*Slave Select*)
		C2OUT	Comparator C2 Output
RE0/AN5	8	RE0	General purpose I/O port E
		AN5	A/D Channel 5
RE1/AN6	9	RE1	General purpose I/O port E
		AN6	A/D Channel 6
RE2/AN7	10	RE2	General purpose I/O port E
		AN7	A/D Channel 7
Vdd	11	+	Positive supply
Vss	12	-	Ground (GND)

Table 1-1 Pin Assignment

Name	Number (DIP 40)	Function	Description
RA7/OSC1/CLKIN	13	RA7	General purpose I/O port A
		OSC1	Crystal Oscillator Input
		CLKIN	External Clock Input
RA6/OSC2/CLKOUT	14	OSC2	Crystal Oscillator Output
		CLKO	Fosc/4 Output
		RA6	General purpose I/O port A
RC0/T1OSO/T1CKI	15	RC0	General purpose I/O port C
		T1OSO	Timer T1 Oscillator Output
		T1CKI	Timer T1 Clock Input
RC1/T1OSO/T1CKI	16	RC1	General purpose I/O port C
		T1OSI	Timer T1 Oscillator Input
		CCP2	CCP1 and PWM1 module I/O
RC2/P1A/CCP1	17	RC2	General purpose I/O port C
		P1A	PWM Module Output
		CCP1	CCP1 and PWM1 module I/O
RC3/SCK/SCL	18	RC3	General purpose I/O port C
		SCK	MSSP module Clock I/O in SPI mode
		SCL	MSSP module Clock I/O in I^2C mode
RD0	19	RD0	General purpose I/O port D
RD1	20	RD1	General purpose I/O port D
RD2	21	RD2	General purpose I/O port D
RD3	22	RD3	General purpose I/O port D
RC4/SDI/SDA	23	RC4	General purpose I/O port A
		SDI	MSSP module *Data* input in SPI mode
		SDA	MSSP module *Data* I/O in I^2C mode

		SDA	MSSP module *Data* I/O in I^2C mode
RC5/SDO	24	RC5	General purpose I/O port C
		SDO	MSSP module *Data* output in SPI mode
RC6/TX/CK	25	RC6	General purpose I/O port C
		TX	USART Asynchronous Output
		CK	USART Synchronous Clock
RC7/RX/DT	26	RC7	General purpose I/O port C
		RX	USART Asynchronous Input
		DT	USART Synchronous Data

Table 1-1 cont. Pin Assignment

Name	Number (DIP 40)	Function	Description
RD4	27	RD4	General purpose I/O port D
RD5/P1B	28	RD5	General purpose I/O port D
		P1B	PWM Output
RD6/P1C	29	RD6	General purpose I/O port D
		P1C	PWM Output
RD7/P1D	30	RD7	General purpose I/O port D
		P1D	PWM Output
Vss	31	-	Ground (GND)
Vdd	32	+	Positive Supply
RB0/AN12/INT	33	RB0	General purpose I/O port B
		AN12	A/D Channel 12
		INT	External Interrupt
RB1/AN10/C12INT3-	34	RB1	General purpose I/O port B
		AN10	A/D Channel 10
		C12INT3-	Comparator C1 or C2 Negative Input
RB2/AN8	35	RB2	General purpose I/O port B
		AN8	A/D Channel 8
RB3/AN9/PGM/C12IN2-	36	RB3	General purpose I/O port B
		AN9	A/D Channel 9
		PGM	Programming enable pin
		C12IN2-	Comparator C1 or C2 Negative Input
RB4/AN11	37	RB4	General purpose I/O port B
		AN11	A/D Channel 11
RB5/AN13/T1G	38	RB5	General purpose I/O port B
		AN13	A/D Channel 13
		T1G	Timer T1 External Input
RB6/ICSPCLK	39	RB6	General purpose I/O port B
		ICSPCLK	Serial programming Clock
RB7			General purpose I/O port B

		ICSPCLK	Serial programming Clock
RB7/ICSPDAT	40	RB7	General purpose I/O port B
		ICSPDAT	Programming enable pin

Table 1-1 cont. Pin Assignment

Central Processor Unit (CPU)

I'm not going to bore you with the operation of the CPU at this stage, however it is important to state that the CPU is manufactured with in RISC technology an important factor when deciding which microprocessor to use.

RISC *Reduced Instruction Set Computer*, gives the PIC16F887 two great advantages:

- The CPU can recognizes only 35 simple instructions (In order to program some other microcontrollers it is necessary to know more than 200 instructions by heart).
- The execution time is the same for all instructions except two and lasts 4 clock cycles (oscillator frequency is stabilized by a quartz crystal). The Jump and Branch instructions execution time is 2 clock cycles. It means that if the microcontroller's operating speed is 20MHz, execution time of each instruc tion will be 200nS, i.e. the program will be executed at the speed of 5 million instructions per second!

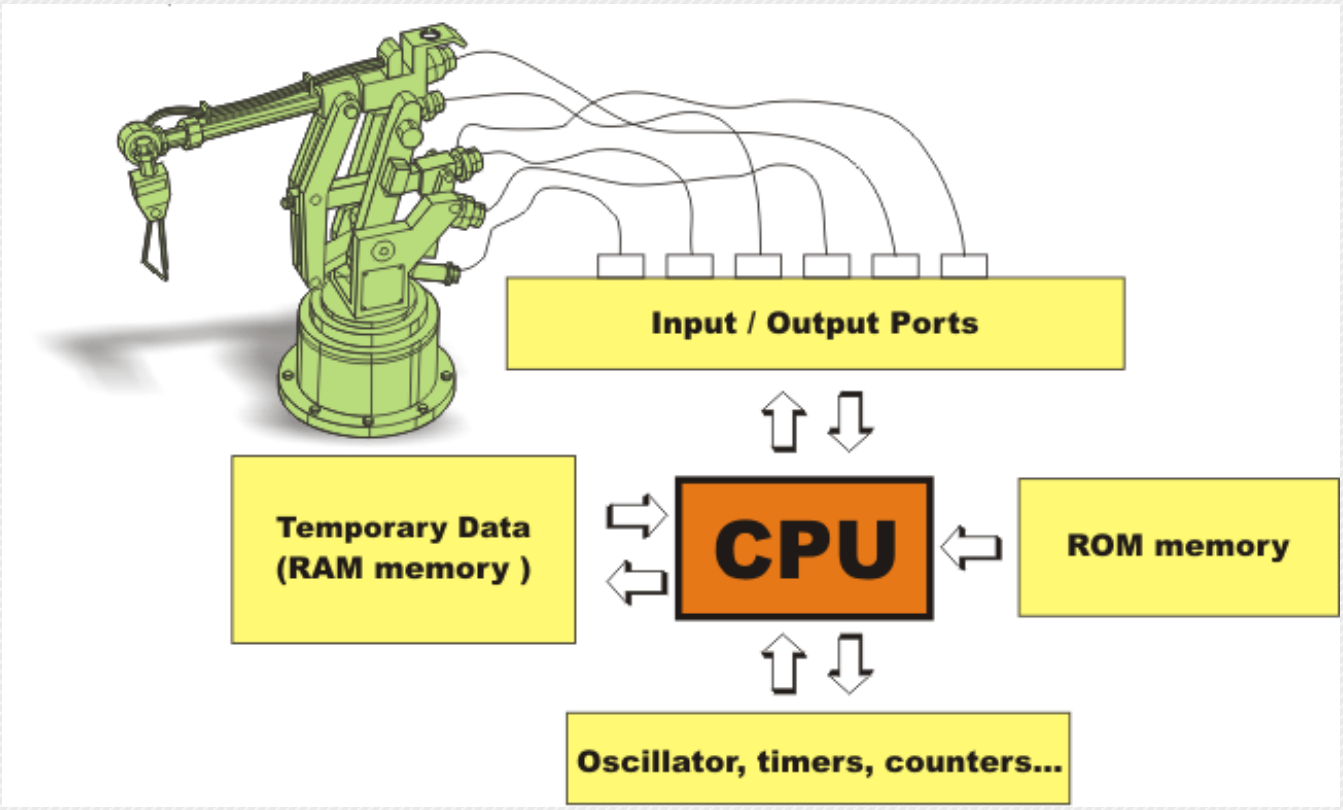


Fig. 1-4 CPU Memory

Memory

This microcontroller has three types of memory- ROM, RAM and EEPROM. All of them will be separately discussed since each has specific functions, features and organization.

ROM Memory

ROM memory is used to permanently save the program being executed. This is why it is often called "program memory". The PIC16F887 has 8Kb of ROM (in total of 8192 locations). Since this ROM is made with FLASH technology, its contents can be changed by providing a special programming voltage (13V).

Anyway, there is no need to explain it in detail because it is automatically performed by means of a special program on

the PC and a simple electronic device called the Programmer.

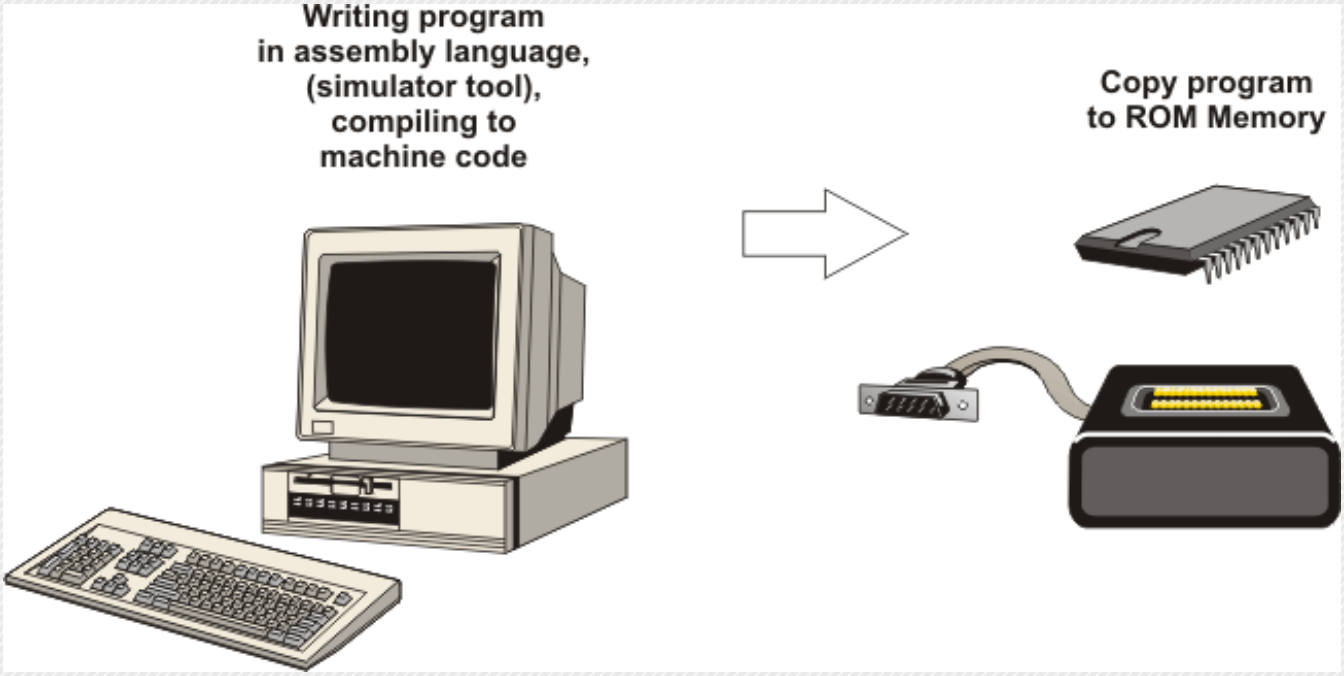


Fig. 1-5 ROM Memory Consept

EEPROM Memory

Similar to program memory, the contents of EEPROM is permanently saved, even the power goes off. However, unlike ROM, the contents of the EEPROM can be changed during operation of the microcontroller. That is why this memory (256 locations) is a perfect one for permanently saving results created and used during the operation.

RAM Memory

This is the third and the most complex part of microcontroller memory. In this case, it consists of two parts: general-purpose registers and special-function registers (SFR).

Even though both groups of registers are cleared when power goes off and even though they are manufactured in the same way and act in the similar way, their functions do not have many things in common.

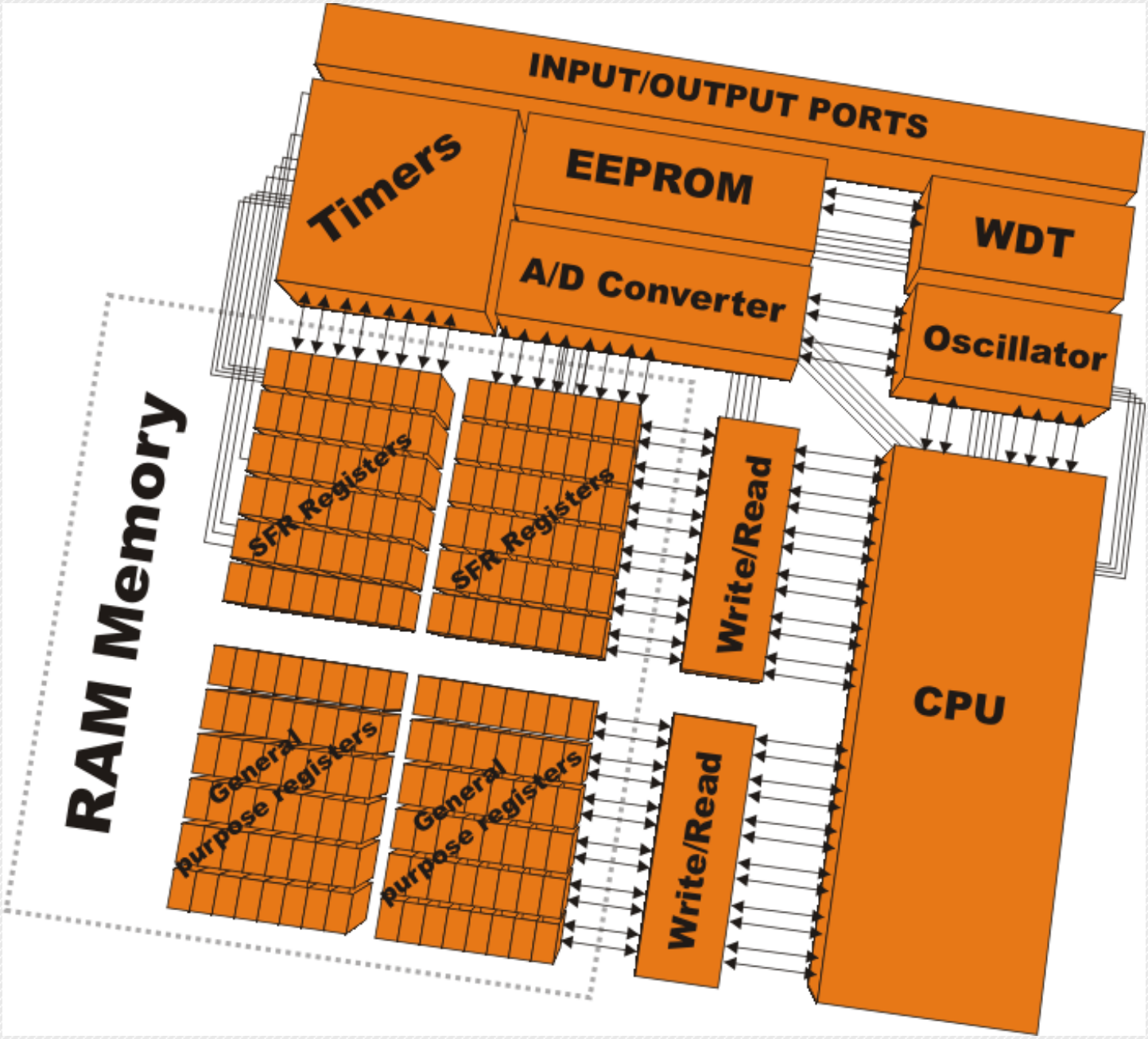


Fig. 1-6 SFR and General Purpose Registers

General-Purpose Registers

General-Purpose registers are used for storing temporary data and results created during operation. For example, if the program performs a counting (for example, counting products on the assembly line), it is necessary to have a register which stands for what we in everyday life call “sum”. Since the microcontroller is not creative at all, it is necessary to specify the address of some general purpose register and assign it a new function. A simple program to increment the value of this register by 1, after each product passes through a sensor, should be created.

Therefore, the microcontroller can execute that program because it now knows what and where the sum which must be incremented is. Similarly to this simple example, each program variable must be preassigned some of general-purpose register.

SFR Registers

Special-Function registers are also RAM memory locations, but unlike general-purpose registers, their purpose is predetermined during manufacturing process and cannot be changed. Since their bits are physically connected to particular circuits on the chip (A/D converter, serial communication module, etc.), any change of their contents directly affects the operation of the microcontroller or some of its circuits. For example, by changing the TRISA register, the

function of each port A pin can be changed in a way it acts as input or output. Another feature of these memory locations is that they have their names (registers and their bits), which considerably facilitates program writing. Since high-level programming language can use the list of all registers with their exact addresses, it is enough to specify the register's name in order to read or change its contents.

RAM Memory Banks

The data memory is partitioned into four banks. Prior to accessing some register during program writing (in order to read or change its contents), it is necessary to select the bank which contains that register. Two bits of the STATUS register are used for bank selecting, which will be discussed later. In order to facilitate operation, the most commonly used SFRs have the same address in all banks which enables them to be easily accessed.

Addr.	Name	Addr.	Name	Addr.	Name	Addr.	Name
00h	INDF	80h	INDF	100h	INDF	180h	INDF
01h	TMR0	81h	OPTION_REG	101h	TMR0	181h	OPTION_REG
02h	PCL	82h	PCL	102h	PCL	182h	PCL
03h	STATUS	83h	STATUS	103h	STATUS	183h	STATUS
04h	FSR	84h	FSR	104h	FSR	184h	FSR
05h	PORTA	85h	TRISA	105h	WDTCON	185h	SRCON
06h	PORTB	86h	TRISB	106h	PORTB	186h	TRISB
07h	PORTC	87h	TRISC	107h	CM1CON0	187h	BAUDCTL
08h	PORTD	88h	TRISD	108h	CM2CON0	188h	ANSEL
09h	PORTE	89h	TRISE	109h	CM2CON1	189h	ANSELH
0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah	PCLATH
0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh	INTCON
0Ch	PIR1	8Ch	PIE1	10Ch	EEDAT	18Ch	EECON1
0Dh	PIR2	8Dh	PIE2	10Dh	EEADR	18Dh	EECON2
0Eh	TMR1L	8Eh	PCON	10Eh	EEDATH	18Eh	Not Used
0Fh	TMR1H	8Fh	OSCCON	10Fh	EEADRH	18Fh	Not Used
10h	T1CON	90h	OSCTUNE	110h	General Purpose Registers 96 bytes	190h	General Purpose Registers 96 bytes
11h	TMR2	91h	SSPCON2				
12h	T2CON	92h	PR2				
13h	SSPBUF	93h	SSPADD				
14h	SSPCON	94h	SSPSTAT				
15h	CCPR1L	95h	WPUB				
16h	CCPR1H	96h	IOCB				
17h	CCP1CON	97h	VRCON				
18h	RCSTA	98h	TXSTA				
19h	TXREG	99h	SPBRG				
1Ah	RCREG	9Ah	SPBRGH				
1Bh	CCPR2L	9Bh	PWM1CON				
1Ch	CCPR2H	9Ch	ECCPAS				
1Dh	CCP2CON	9Dh	PSTRCON				
1Eh	ADRESH	9Eh	ADRESL				
1Fh	ADCON0	9Fh	ADCON1				
20h	General Purpose Registers 96 bytes	A0h	General Purpose Registers 80 bytes				
7Fh		FFh		17Fh		1EFh	
	Bank 0		**Bank 1**		**Bank 2**		**Bank 3**

Table 1-2 Address Banks

SFRs bank 0

Address	Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
00h	INDF	Indirect register							
01h	TMR0	Timer T0 Register							
02h	PCL	Least Significant Byte of Program Counter							
03h	STATUS	IRP	RP1	RP0	TO	PD	Z	DC	C
04h	FSR	Indirect Data Memory Address Pointer							
05h	PORTA	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
07h	PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0
08h	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0
09h	PORTE	-	-	-	-	RE3	RE2	RE1	RE0
0Ah	PCLATH	-	-	-	Upper 5 bits of Program Counter				
0Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
0Ch	PIR1	-	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
0Dh	PIR2	OSFIF	C2IF	C1IF	EEIF	BCLIF	ULPWUIF	-	CCP2IF
0Eh	TMR1L	Least Significant Byte of the 16-bit Timer TMR0							
0Fh	TMR1H	Most Significant Byte of the 16-bit Timer TMR0							
10h	T1CON	T1GINV	TMR1GE	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON
11h	TMR2	Timer T2 Register							
12h	T2CON	-	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
13h	SSPBUF	Synchronous Serial Port Receive Buffer/Transmit Register							
14h	SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
15h	CCPR1L	Capture/ComparePWM Register 1 Low Byte (LSB)							
16h	CCPR1H	Capture/ComparePWM Register 1 High Byte (LSB)							
17h	CCP1CON	P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0
18h	RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
19h	TXREG	EUSART Transmit Data Register							
1Ah	RCREG	EUSART Receive Data Register							
1Bh	CCPR2L	Capture/Compare PWM Register 1 Low Byte (LSB)							
1Ch	CCPR2H	Capture/Compare PWM Register 1 High Byte (LSB)							
1Dh	CCP2CON	-	-	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0
1Eh	ADRESH	A/D Result Register High Byte							
1Fh	ADCON0	ADCS1	ADCS0	CHS3	CHS2	CHS1	CHS0	GO/DONE	ADON

Table 1-3 SFR Bank 0

SFRs bank 1

Address	Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
80h	INDF	Indirect Register							
81h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
82h	PCL	Least Significant Byte of Program Counter							
83h	STATUS	IRP	RP1	RP0	TO	PD	Z	DC	C
84h	FSR	Indirect Data Memory Address Pointer							
85h	TRISA	TRISA7	TRISA6	TRISA5	TRISA4	TRISA3	TRISA2	TRISA1	TRISA0
86h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0
87h	TRISC	TRISC7	TRISC6	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0
88h	TRISD	TRISD7	TRISD6	TRISD5	TRISD4	TRISD3	TRISD2	TRISD1	TRISD0
89h	TRISE	-	-	-	-	TRISE3	TRISE2	TRISE1	TRISE0
8Ah	PCLATH	-	-	-	Upper 5 bits of the Ptogram Counter				
8Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
8Ch	PIE1	-	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
8Dh	PIE2	OSFIE	C2IE	C1IE	EEIE	BCLIE	ULPWUIE	-	CCP2IE
8Eh	PCON	-	-	ULPWUE	SBOREN	-	-	POR	BOR
8Fh	OSCCON	-	IRCF2	IRCF1	IRCF0	OSTS	HTS	LTS	SCS
90h	OSCTUNE	-	-	-	TUN4	TUN3	TUN2	TUN1	TUN0
91h	SSPCON2	GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN
92h	PR2	Timer T2 Period Register							
93h	SSPADD	Synchronous Serial Port (I^2C mode) Address Register							
93h	SSPMSK	MSK7	MSK6	MSK5	MSK4	MSK3	MSK2	MSK1	MSK0
94h	SSPSTAT	SMP	CKE	D/A	P	S	R/W	UA	BF
95h	WPUB	WPUB7	WPUB6	WPUB5	WPUB4	WPUB3	WPUB2	WPUB1	WPUB0
96h	IOCB	IOCB7	IOCB6	IOCB5	IOCB4	IOCB3	IOCB2	IOCB1	IOCB0
97h	VRCON	VREN	VROE	VRR	VRSS	VR3	VR2	VR1	VR0
98h	TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D
99h	SPBRG	BRG7	BRG6	BRG5	BRG4	BRG3	BRG2	BRG1	BRG0
9Ah	SPBRGH	BRG15	BRG14	BRG13	BRG12	BRG11	BRG10	BRG9	BRG8
9Bh	PWM1CON	PRSEN	PDC6	PDC5	PDC4	PDC3	PDC2	PDC1	PDC0
9Ch	ECCPAS	ECCPASE	ECCPAS2	ECCPAS1	ECCPAS0	PSSAC1	PSSAC0	PSSBD1	PSSBD0
9Dh	PSTRCON	-	-	-	STRSYNC	STRD	STRC	STRB	STRA
9Eh	ADRESL	A/D Result Register Low Byte							
9Fh	ADCON1	ADFM	-	VCFG1	VCFG0	-	-	-	-

Table 1-4 SFR Bank 1

SFRs bank 2

Address	Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
100h	INDF	Indirect register							
101h	TMR0	Timer T0 Register							
102h	PCL	Least Significant Byte of the Program Counter							
103h	STATUS	IRP	RP1	RP0	TO	PD	Z	DC	C
104h	FSR	Indirect Data Memory Address Pointer							
105h	WDTCON	-	-	-	WDTPS3	WDTPS2	WDTPS1	WDTPS0	SWDTEN
106h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
107h	CM1CON0	C1ON	C1OUT	C1OE	C1POL	-	C1R	C1CH1	C1CH0
108h	CM2CON0	C2ON	C2OUT	C2OE	C2POL	-	C2R	C2CH1	C2CH0
109h	CM2CON1	MC1OUT	MC2OUT	C1RSEL	C2RSEL	-	-	T1GSS	C2SYNC
10Ah	PCLATH	-	-	-	Upper 5 bits of the Program Counter				
10Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
10Ch	EEDAT	EEDAT7	EEDAT6	EEDAT5	EED AT4	EEDAT3	EEDAT2	EEDAT1	EEDAT0
10Dh	EEADR	EEADR7	EEADR6	EEADR5	EEADR4	EEADR3	EEADR2	EEADR1	EEADR0
10Eh	EEDATH	-	-	EEDATH5	EEDATH4	EEDATH3	EEDATH2	EEDATH1	EEDATH0
10Fh	EEADRH	-	-	-	EEADRH4	EEADRH3	EEADRH2	EEADRH1	EEADRH0

Table 1-5 SFR Bank 2

SFRs bank 3

Address	Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
180h	INDF	Indirect Register							
181h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
182h	PCL	Least Significan Byte of the Program Counter							
183h	STATUS	IRP	RP1	RP0	TO	PD	Z	DC	C
184h	FSR	Indirect Data Memory Address Pointer							
185h	SRCON	SR1	SR0	C1SEN	C2REN	PULSS	PULSR	-	FVREN
186h	TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0
187h	BAUDCTL	ABDOVF	RCIDL	-	SCKP	BRG16	-	WUE	ABDEN
188h	ANSEL	ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0
189h	ANSELH	-	-	ANS13	ANS12	ANS11	ANS10	ANS9	ANS8
19Ah	PCLATH	-	-	-	Upper 5 bits of the Program Counter				
19Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
19Ch	EECON1	EEPGD	-	-	-	WRERR	WREN	WR	RD
19Dh	EECON2	EEPROM Control Register 2							

Table 1-6 SFR Bank 3

STACK

A part of the RAM used for the stack consists of eight 13-bit registers. Before the microcontroller starts to execute a subroutine (CALL instruction) or when an interrupt occurs, the address of first next instruction being currently executed is pushed onto the stack, i.e. onto one of its registers. In that way, upon subroutine or interrupt execution, the microcontroller knows from where to continue regular program execution. This address is cleared upon return to the main program because there is no need to save it any longer, and one location of the stack is automatically available for further use.

It is important to understand that data is always circularly pushed onto the stack. It means that after the stack has been

pushed eight times, the ninth push overwrites the value that was stored with the first push. The tenth push overwrites the second push and so on. Data overwritten in this way is not recoverable. In addition, the programmer cannot access these registers for write or read and there is no Status bit to indicate stack overflow or stack underflow conditions. For that reason, one should take special care of it during program writing.

Interrupt System

The first thing that the microcontroller does when an interrupt request arrives is to execute the current instruction and then stop regular program execution. Immediately after that, the current program memory address is automatically pushed onto the stack and the default address (predefined by the manufacturer) is written to the program counter. That location from where the program continues execution is called the interrupt vector. For the PIC16F887 microcontroller, this address is 0004h. As seen in Fig. 1-7 below, the location containing interrupt vector is passed over during regular program execution.

Part of the program being activated when an interrupt request arrives is called the interrupt routine. Its first instruction is located at the interrupt vector. How long this subroutine will be and what it will be like depends on the skills of the programmer as well as the interrupt source itself. Some microcontrollers have more interrupt vectors (every interrupt request has its vector), but in this case there is only one. Consequently, the first part of the interrupt routine consists in interrupt source recognition.

Finally, when the interrupt source is recognized and interrupt routine is executed, the microcontroller reaches the `RETFIE` instruction, pops the address from the stack and continues program execution from where it left off.

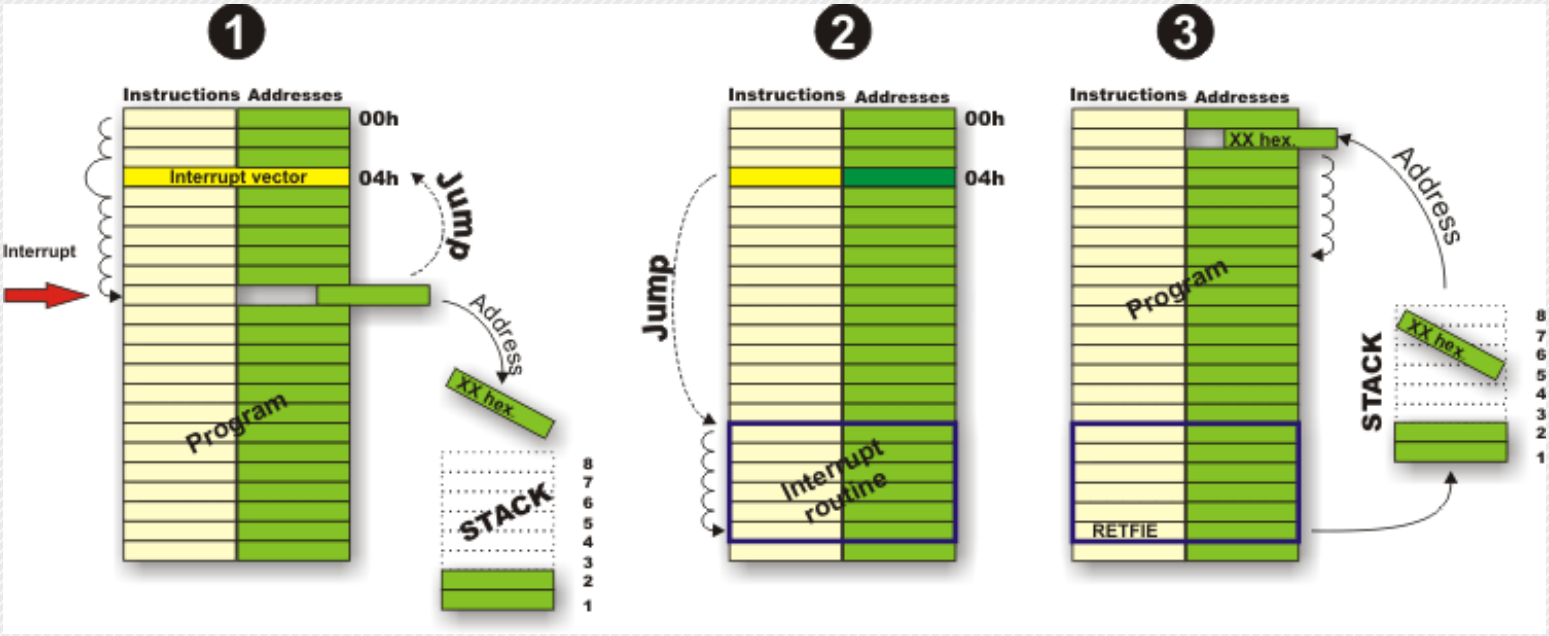


Fig.1-7 Interrupt System

How to use SFRs

You have bought the microcontroller and have a good idea how to use it...There is a long list of SFRs with all bits. Each of them controls some process. All in all, it looks like a big control table with a lot of instruments and switches. Now you are concerned about whether you will manage to learn how to use them all? You will probably not, but don't worry, you don't have to! Such powerful microcontrollers are similar to a supermarkets: they offer so many things at low prices and it is only up to you to choose. Therefore, select the field you are interested in and study only what you need to know. Afterwards, when you completely understand hardware operation, study SFRs which are in control of it (there are usually a few of them). To reiterate, during program writing and prior to changing some bits of these registers, do not forget to select the appropriate bank. This is why they are listed in the tables above.

Previous Chapter | Table of Contents | Next Chapter

Chapter 2: Core SFRs

Features and Function

The special function registers can be classified into two categories:

- Core (CPU) registers - control and monitor operation and processes in the central processor. Even though there are only a few of them, the operation of the whole microcontroller depends on their contents.
- Peripheral SFRs- control the operation of peripheral units (serial communication module, A/D converter etc.). Each of these registers is mainly specialized for one circuit and for that reason they will be described along with the circuit they are in control of.

The core (CPU) registers of the PIC16F887 microcontroller are described in this chapter. Since their bits control several different circuits within the chip, it is not possible to classify them into some special group. These bits are described along with the processes they control.

STATUS Register

	R/W (0)	R/W (0)	R/W (0)	R (1)	R (1)	R/W (x)	R/W (x)	R/W (x)	Features
STATUS	IRP	RP1	RP0	TO	PD	Z	DC	C	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend	
R/W	Readable/Writable bit
R	Readable bit only
(0)	After reset, bit is cleared
(1)	After reset, bit is set
(x)	After reset, bit is unknown

Fig. 2-1 STATUS Register

The STATUS register contains: the arithmetic status of the W register, the RESET status and the bank select bits for data memory. One should be careful when writing a value to this register because if you do it wrong, the results may be different than expected. For example, if you try to clear all bits using the `CLRF STATUS` instruction, the result in the register will be 000xx1xx instead of the expected 00000000. Such errors occur because some of the bits of this register are set or cleared according to the hardware as well as because the bits 3 and 4 are readable only. For these reasons, if it is required to change its content (for example, to change active bank), it is recommended to use only instructions which do

not affect any Status bits (C, DC and Z). Refer to “Instruction Set Summary”.

- **IRP** - Bit selects register bank. It is used for indirect addressing.
  - **1** - Banks 0 and 1 are active (memory location 00h-FFh)
  - **0** - Banks 2 and 3 are active (memory location 100h-1FFh)

- **RP1,RP0** - Bits select register bank. They are used for direct addressing.

RP1	RP0	Active Bank
0	0	Bank0
0	1	Bank1
1	0	Bank2
1	1	Bank3

Table 2-1

- **TO - Time-out bit.**
  - **1** - After power-on or after executing `CLRWDT` instruction which resets watch-dog timer or SLEEP instruction which sets the microcontroller into low-consumption mode.
  - **0** - After watch-dog timer time-out has occurred.

- **PD - Power-down bit.**
  - **1** - After power-on or after executing `CLRWDT` instruction which resets watch-dog timer.
  - **0** - After executing `SLEEP` instruction which sets the microcontroller into low-consumption mode.

- **Z - Zero bit**
  - **1** - The result of an arithmetic or logic operation is zero.
  - **0** - The result of an arithmetic or logic operation is different from zero.

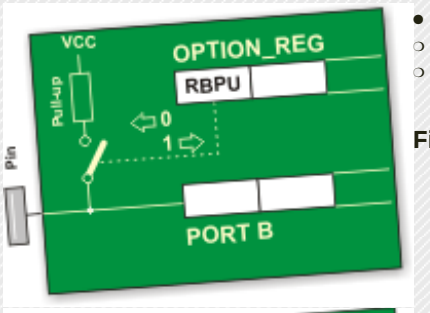
- **DC - Digit carry/borrow** bit is changed during addition and subtraction if an “overflow” or a “borrow” of the result occurs.
  - **1** - A carry-out from the 4th low-order bit of the result has occurred.
  - **0** - No carry-out from the 4th low-order bit of the result has occurred.

- **C - Carry/Borrow bit** is changed during addition and subtraction if an “overflow” or a “borrow” of the result occurs, i.e. if the result is greater than 255 or less than 0.
  - **1** - A carry-out from the most significant bit of the result has occurred.
  - **0** - No carry-out from the most significant bit of the result has occurred.

OPTION_REG Register

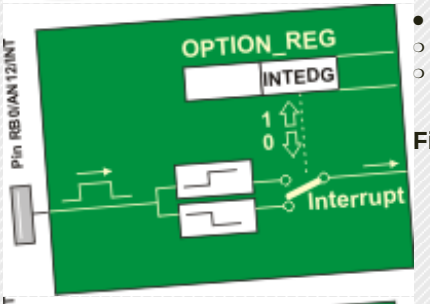
Fig.2-2

The OPTION_REG register contains various control bits to configure: Timer0/WDT prescaler, timer TMR0, external interrupt and pull-ups on PORTB.



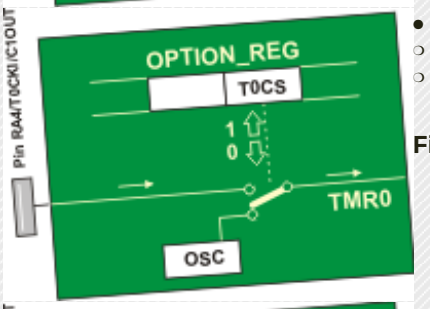
- **RBPU - Port B Pull up Enable bit.**
  - **1** - PortB pull-ups are disabled.
  - **0** - PortB pull-ups are enabled.

Fig.2-3



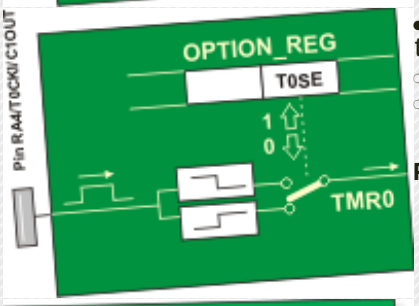
- **INTEDG - Interrupt Edge Select bit.**
  - **1** - Interrupt on rising edge of RB0/INT pin.
  - **0** - Interrupt on falling edge of RB0/INT pin.

Fig.2-4



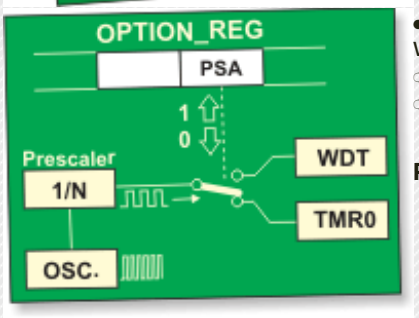
- **T0CS - TMR0 Clock Source Select bit.**
  - **1** - Transition on TOCKI pin.
  - **0** - Internal instruction cycle clock (Fosc/4).

Fig.2-5



- **T0SE - TMR0 Source Edge Select bit** selects pulse edge (rising or falling) counted by the timer TMR0 through the RA4/T0CKI pin.
  - **1** - Increment on high-to-low transition on TOCKI pin.
  - **0** - Increment on low-to-high transition on TOCKI pin.

Fig.2-6



- **PSA - Prescaler Assignment bit** assigns prescaler (only one exists) to the timer or watchdog timer.
  - **1** - Prescaler is assigned to the WDT.
  - **0** - Prescaler is assigned to the TMR0.

Fig.2-7

PS2, PS1, PS0 Prescaler Rate Select bits

Prescaler rate is selected by combining these three bits. Described, as shown in the table below, prescaler rate depends on whether prescaler is assigned (TMR0) or watch-dog timer (WDT).

PS2	PS1	PS0	TMR0	WDT
0	0	0	1:2	1:1
0	0	1	1:4	1:2
0	1	0	1:8	1:4
0	1	1	1:16	1:8
1	0	1	1:64	1:32

1	1	0	1:128	1:64
1	1	1	1:256	1:128

Table 2-2

In order to achieve 1:1 prescaler rate when the timer TMR0 counts up pulses, the prescaler should be assigned to the WDT. As a result of this, the timer TMR0 does not use the prescaler, but directly counts pulses generated by the oscillator, which was the objective!

Interrupt System Registers

When an interrupt request arrives it does not mean that interrupt will automatically occur, because it must also be enabled by the user (from within the program). Because of that, there are special bits used to enable or disable interrupts. It is easy to recognize these bits by IE contained in their names (stands for Interrupt Enable). Besides, each interrupt is associated with another bit called the flag which indicates that interrupt request has arrived regardless of whether it is enabled or not. They are also easily recognizable by the last two letters contained in their names- IF (Interrupt Flag).

As seen, everything is based on a simple and efficient idea. When an interrupt request arrives, the flag bit is to be set first.

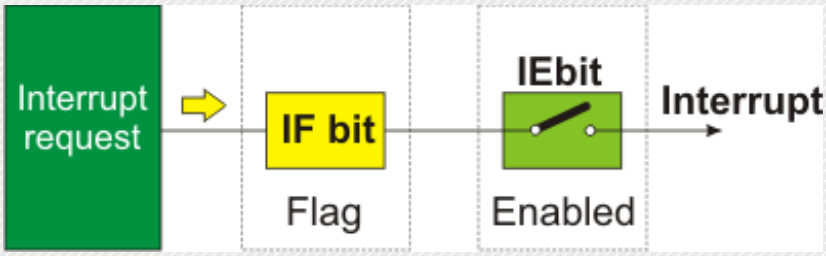


Fig. 2-8 Interrupt System Registers

If the appropriate IE bit is not set (0), this event will be completely ignored. Otherwise, an interrupt occurs! In case several interrupt sources are enabled, it is necessary to detect the active one before the interrupt routine starts execution. Source detection is performed by checking flag bits.

It is important to understand that the flag bits are not automatically cleared, but by software during interrupt routine execution. If this detail is neglected, another interrupt will occur immediately upon return to the program, even though there are no more requests for its execution! Simply put, the flag as well as IE bit remained set.

All interrupt sources typical of the PIC16F887 microcontroller are shown on the next page. Note several things:

- **GIE bit** - enables all unmasked interrupts and disables all interrupts simultaneously.
- **PEIE bit** - enables all unmasked peripheral interrupts and disables all peripheral interrupts (This does not concern Timer TMR0 and port B interrupt sources).

To enable interrupt caused by changing logic state on port B, it is necessary to enable it for each bit separately. In this case, bits of the **IOCB** register have the function to control IE bits.

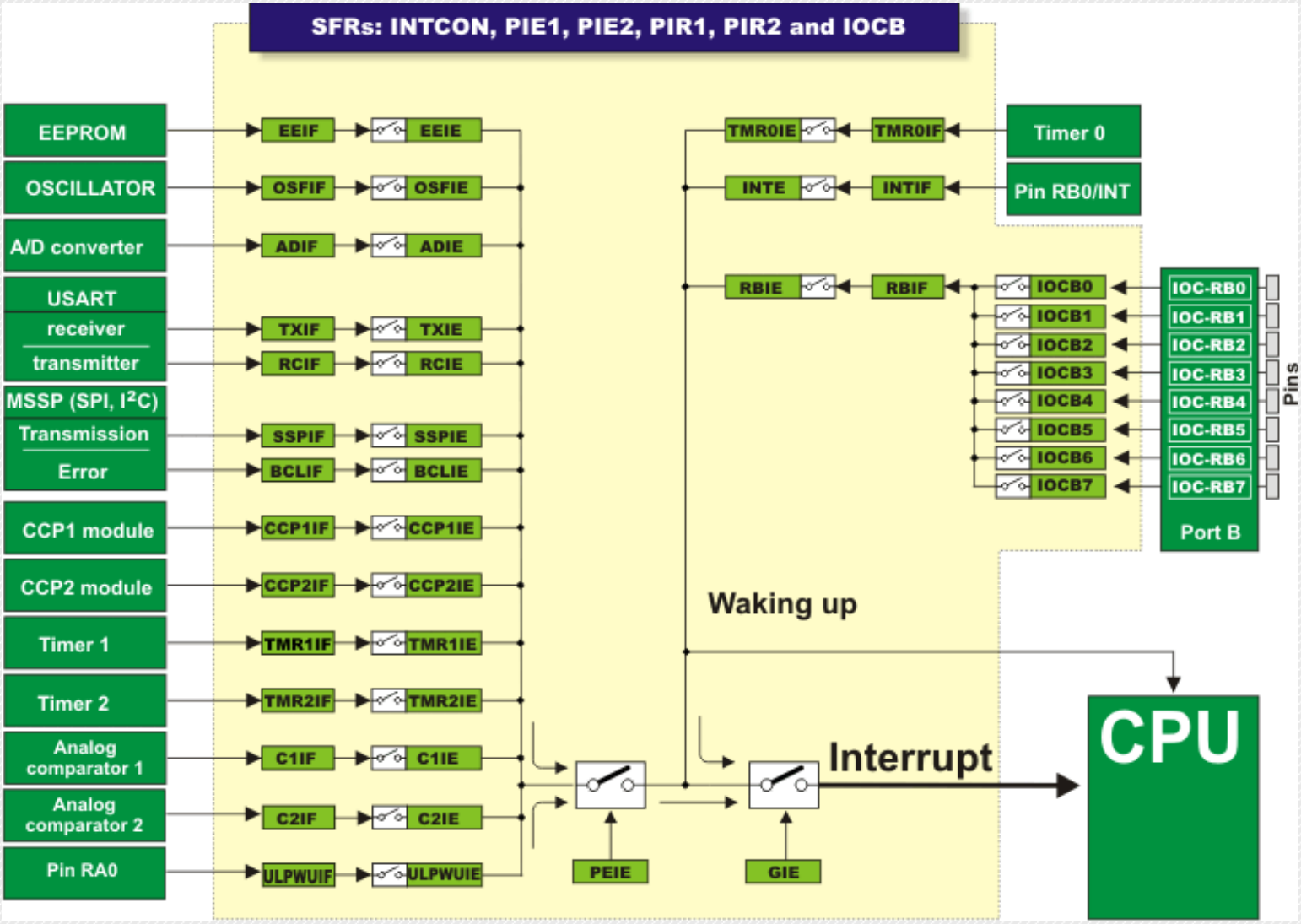


Fig. 2-9 Interrupt SFRs

INTCON Register

The INTCON register contains various enable and flag bits for TMR0 register overflow, PORTB change and external INT pin interrupts.

	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (x)	Features
INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

R/W	Readable/Writable bit
(0)	After reset, bit is cleared
(x)	After reset, bit is unknown

Fig. 2-10 INTCON Register

- **GIE - Global Interrupt Enable bit** - controls all possible interrupt sources simultaneously.
  - **1** - Enables all unmasked interrupts.
  - **0** - Disables all interrupts.

- **PEIE - Peripheral Interrupt Enable bit** acts similar to GIE, but controls interrupts enabled by peripherals. It means

that it does not affect interrupts triggered by the timer TMR0 or by changing state on port B or RB0/INT pin.
  - **1** - Enables all unmasked peripheral interrupts.
  - **0** - Disables all peripheral interrupts.

- **T0IE - TMR0 Overflow Interrupt Enable bit** controls interrupt enabled by TMR0 overflow.
  - **1** - Enables the TMR0 interrupt.
  - **0** - Disables the TMR0 interrupt.

- **INTE - RB0/INT External Interrupt Enable bit** controls interrupt caused by changing logic state on pin RB0/IN (external interrupt).
  - **1** - Enables the INT external interrupt.
  - **0** - Disables the INT external interrupt.

- **RBIE - RB Port Change Interrupt Enable bit**. When configured as inputs, port B pins may cause interrupt by changing their logic state (no matter whether it is highto- low transition or vice versa, fact that something is changed only matters). This bit determines whether interrupt is to occur or not.
  - **1** - Enables the port B change interrupt.
  - **0** - Disables the port B change interrupt.

- **T0IF - TMR0 Overflow Interrupt Flag bit** registers the timer TMR0 register overflow, when counting starts from zero.
  - **1** - TMR0 register has overflowed (bit must be cleared in software).
  - **0** - TMR0 register has not overflowed.

- **INTF - RB0/INT External Interrupt Flag bit** registers change of logic state on the RB0/INT pin.
  - **1** - The INT external interrupt has occurred (must be cleared in software).
  - **0** - The INT external interrupt has not occurred.

- **RBIF - RB Port Change Interrupt Flag bit** registers change of logic state of some port B input pins.
  - **1** - At least one of the port B general purpose I/O pins has changed state. Upon reading portB, RBIF (flag bit) must be cleared in software.
  - **0** - None of the port B general purpose I/O pins has changed state.

PIE1 Register

The PIE1 register contains the peripheral interrupt enable bits.

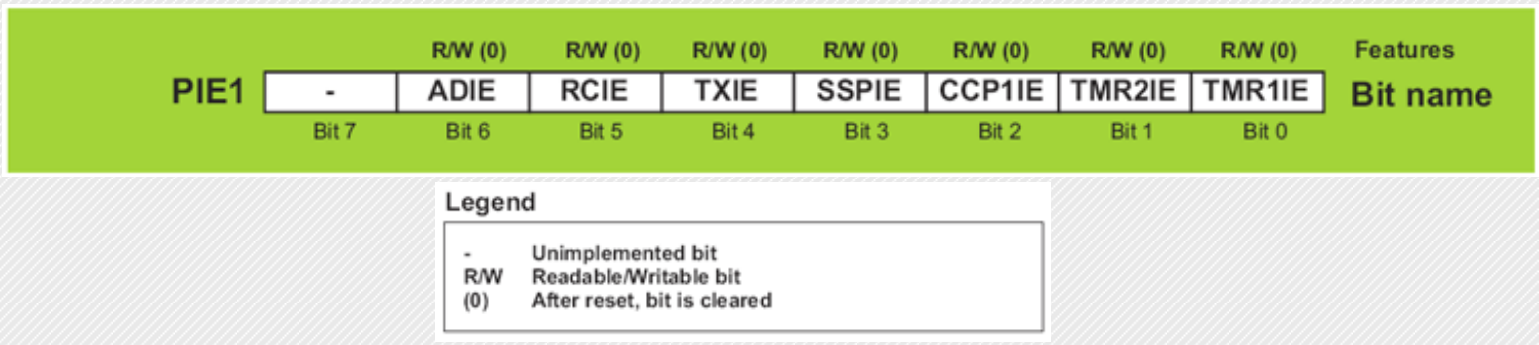


Fig. 2-11 PIE1 register

- **ADIE - A/D Converter Interrupt Enable bit.**
  - **1** - Enables the ADC interrupt.
  - **0** - Disables the ADC interrupt.

- **RCIE - EUSART Receive Interrupt Enable bit.**
  - **1** - Enables the EUSART receive interrupt.
  - **0** - Disables the EUSART receive interrupt.

- **TXIE - EUSART Transmit Interrupt Enable bit.**
  - **1** - Enables the EUSART transmit interrupt.
  - **0** - Disables the EUSART transmit interrupt.

- **SSPIE - Master Synchronous Serial Port (MSSP) Interrupt Enable bit** - enables an interrupt request to be generated after each data transfer via synchronous serial communication module (SPI or I2C mode).
  - **1** - Enables the MSSP interrupt.
  - **0** - Disables the MSSP interrupt.

- **CCP1IE - CCP1 Interrupt Enable bit** enables an interrupt request to be generated in CCP1 module used for PWM signal processing.
  - **1** - Enables the CCP1 interrupt.
  - **0** - Disables the CCP1 interrupt.

- **TMR2IE - TMR2 to PR2 Match Interrupt Enable bit**
  - **1** - Enables the TMR2 to PR2 match interrupt.
  - **0** - Disables the TMR2 to PR2 match interrupt.

- **TMR1IE - TMR1 Overflow Interrupt Enable bit** enables an interrupt request to be generated after each timer TMR1 register overflow, i.e. when the counting starts from zero.
  - **1** - Enables the TMR1 overflow interrupt.
  - **0** - Disables the TMR1 overflow interrupt.

PIE2 Register

The PIE2 Register also contains the various interrupt enable bits.

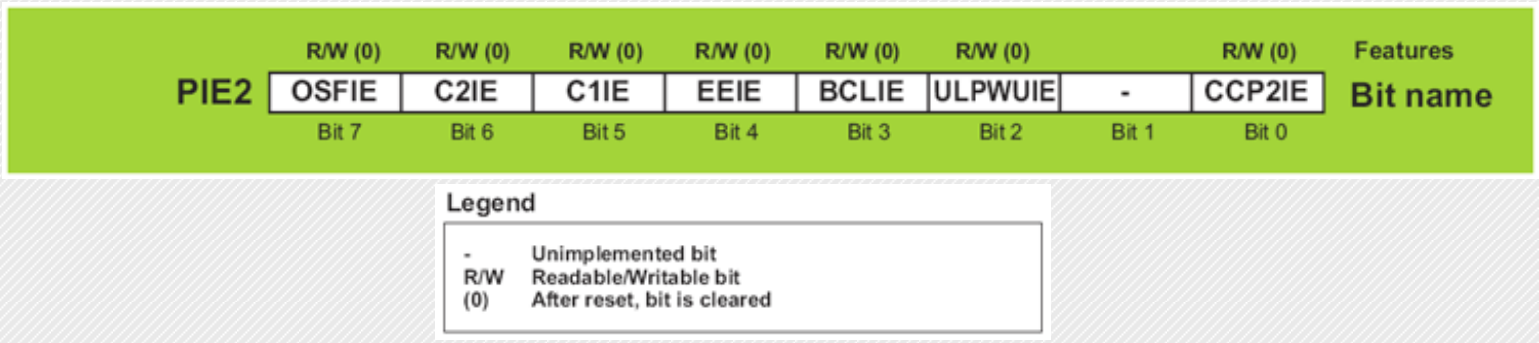


Fig. 2-12 PIE2 Register

- **OSFIE - Oscillator Fail Interrupt Enable bit.**
  - **1** - Enables oscillator fail interrupt.
  - **0** - Disables oscillator fail interrupt.

- **C2IE - Comparator C2 Interrupt Enable bit.**
  - **1** - Enables Comparator C2 interrupt.
  - **0** - Disables Comparator C2 interrupt.

- **C1IE - Comparator C1 Interrupt Enable bit.**
  - **1** - Enables Comparator C1 interrupt.
  - **0** - Disables Comparator C1 interrupt.

- **EEIE - EEPROM Write Operation Interrupt Enable bit.**
  - **1** - Enables EEPROM write operation interrupt.
  - **0** - Disables EEPROM write operation interrupt.

- **BCLIE - Bus Collision Interrupt Enable bit.**
  - **1** - Enables bus collision interrupt.
  - **0** - Disables bus collision interrupt.

- **ULPWUIE - Ultra Low-Power Wake-up Interrupt Enable bit.**
  - **1** - Enables Ultra Low-Power Wake-up interrupt.
  - **0** - Disables Ultra Low-Power Wake-up interrupt.

- **CCP2IE - CCP2 Interrupt Enable bit.**

- **1** - Enables CCP2 interrupt.
- **0** - Disables CCP2 interrupt.

PIR1 Register

The PIR1 register contains the interrupt flag bits.

		R/W (0)	R (0)	R (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	Features
PIR1	-	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend	
-	Unimplemented bit
R/W	Readable/Writable bit
R	Readable bit
(0)	After reset, bit is cleared

Fig. 2-13 PIR1 Register

- **ADIF - A/D Converter Interrupt Flag bit.**
  - **1** - A/D conversion is completed (bit must be cleared in software).
  - **0** - A/D conversion is not completed or has not started.

- **RCIF - EUSART Receive Interrupt Flag bit.**
  - **1** - The EUSART receive buffer is full. Bit is cleared by reading the RCREG register.
  - **0** - The EUSART receive buffer is not full.

- **TXIF - EUSART Transmit Interrupt Flag bit.**
  - **1** - The EUSART transmit buffer is empty. Bit is cleared by writing to the TXREG register.
  - **0** - The EUSART transmit buffer is full.

- **SSPIF - Master Synchronous Serial Port (MSSP) Interrupt Flag bit.**
  - **1** - The MSSP interrupt condition during data transmit/receive has occurred. These conditions differ depending on MSSP operating mode (SPI or I2C) This bit must be cleared in software before returning from the interrupt service routine.
  - **0** - No MSSP interrupt condition has occurred.

- **CCP1IF - CCP1 Interrupt Flag bit.**
  - **1** - CCP1 interrupt condition has occurred (CCP1 is unit for capturing, comparing and generating PWM signal). Depending on operating mode, capture or compare match has occurred. In both cases, bit must be cleared in software. This bit is not used in PWM mode.
  - **0** - No CCP1 interrupt condition has occurred.

- **TMR2IF - Timer2 to PR2 Interrupt Flag bit**
  - **1** - TMR2 (8-bit register) to PR2 match has occurred. This bit must be cleared in software before returning from the interrupt service routine.
  - **0** - No TMR2 to PR2 match has occurred.

- **TMR1IF - Timer1 Overflow Interrupt Flag bit**
  - **1** - The TMR1 register has overflowed. This bit must be cleared in software.
  - **0** - The TMR1 register has not overflowed.

PIR2 Register

The PIR2 register contains the interrupt flag bits.

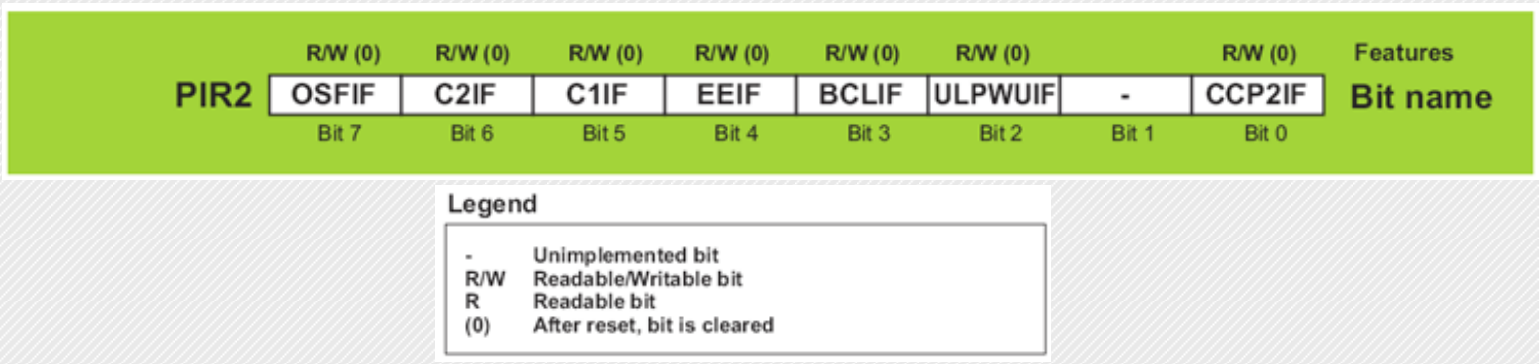


Fig. 2-14 PIR2 register

- **OSFIF - Oscillator Fail Interrupt Flag bit.**
  - **1** - System oscillator failed and clock input has changed to internal oscillator INTOSC. This bit must be cleared in software.
  - **0** - System oscillator operates normally.

- **C2IF - Comparator C2 Interrupt Flag bit.**
  - **1** - Comparator C2 output has changed (bit C2OUT). This bit must be cleared in software.
  - **0** - Comparator C2 output has not changed.

- **C1IF - Comparator C1 Interrupt Flag bit.**
  - **1** - Comparator C1 output has changed (bit C1OUT). This bit must be cleared in software.
  - **0** - Comparator C1 output has not changed.

- **EEIF - EE Write Operation Interrupt Flag bit.**
  - **1** - EEPROM write completed. This bit must be cleared in software.
  - **0** - EEPROM write is not completed or has not started.

- **BCLIF - Bus Collision Interrupt Flag bit.**
  - **1** - A bus collision has occurred in the MSSP when configured for I2C Master mode. This bit must be cleared in software.
  - **0** - No bus collision has occurred.

- **ULPWUIF - Ultra Low-power Wake-up Interrupt Flag bit.**
  - **1** - Wake-up condition has occurred. This bit must be cleared in software.
  - **0** - No Wake-up condition has occurred.

- **CCP2IF - CCP2 Interrupt Flag bit.**
  - **1** - CCP2 interrupt condition has occurred (unit for capturing, comparing and generating PWM signal). Depending on operating mode, capture or compare match has occurred. In both cases, the bit must be cleared in software. This bit is not used in PWM mode.
  - **0** - No CCP2 interrupt condition has occurred.

PCON register

The PCON register contains only two flag bits used to differentiate between a: power-on reset, brown-out reset, Watchdog Timer Reset and external reset (through MCLR pin).

Fig. 2-15 PCON register

- **ULPWUE - Ultra Low-Power Wake-up Enable bit**
  - **1** - Ultra Low-Power Wake-up enabled.
  - **0** - Ultra Low-Power Wake-up disabled.

- **SBOREN - Software BOR Enable bit**
  - **1** - Brown-out Reset enabled.
  - **0** - Brown-out Reset disabled.

- **POR - Power-on Reset Status bit**
  - **1** - No Power-on reset has occurred.
  - **0** - Power-on reset has occurred. This bit must be set in software after a Power-on Reset occurs.

- **BOR - Brown-out Reset Status bit**
  - **1** - No Brown-out reset has occurred.
  - **0** - Brown-out reset has occurred. This bit must be set in software after a Brown-out Reset occurs.

PCL and PCLATH Registers

The size of the program memory of the PIC16F887 is 8K. Therefore, it has 8192 locations for program storing. For this reason the program counter must be 13-bits wide (2^13 = 8192). In order that the contents of some location may be changed in software during operation, its address must be accessible through some SFR. Since all SFRs are 8-bits wide, this register is “artificially” created by dividing its 13 bits into two independent registers: PCLATH and PCL.

If the program execution does not affect the program counter, the value of this register is automatically and constantly incremented +1, +1, +1, +1... In that way, the program is executed just as it is written- instruction by instruction, followed by a constant address increment.

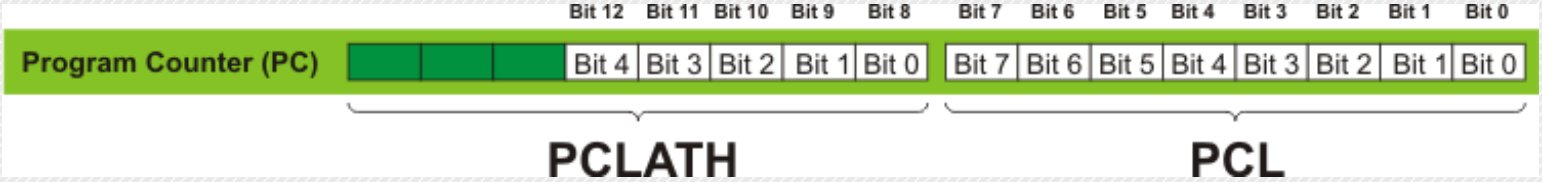


Fig. 2-16 PCL and PCLATH Registers

If the program counter is changed in software, then there are several things that should be kept in mind in order to avoid problems:

- Eight lower bits (the low byte) come from the PCL register which is readable and writable, whereas five upper bits coming from the PCLATH register are writable only.
- The PCLATH register is cleared on any reset.
- In assembly language, the value of the program counter is marked with PCL, but it obviously refers to 8 lower bits only. One should take care when using the “`ADDWF PCL`” instruction. This is a jump instruction which specifies the target location by adding some number to the current address. It is often used when jumping into a look-up table or program branch table to read them. A problem arises if the current address is such that addition causes change on some bit belonging to the higher byte of the PCLATH register. Do you see what is going on?

  Executing any instruction upon the PCL register simultaneously causes the Prog ram Counter bits to be replaced by the contents of the PCLATH register. However, the PCL register has access to only 8 lower bits of the instruction result and the following jump will be completely incorrect. The problem is solved by setting such instructions at addresses ending by xx00h. This enables the program to jump up to 255 locations. If longer jumps are executed by this instruction, the PCLATH register must be incremented by 1 for each PCL register overflow.

- On subroutine call or jump execution (instructions `CALL` and `GOTO`), the microcontroller is able to provide only 11-bit addressing. For this reason, similar to RAM which is divided in “banks”, ROM is divided in four “pages” in size of 2K each. Such instructions are executed within these pages without any problems. Simply, since the processor is provided with 11-bit address from the program, it is able to address any location within 2KB. Figure 2-17 below illustrates this situation as a jump to the subroutine PP1 address.

  However, if a subroutine or jump address are not within the same page as the location from where the jump is, two “missing”- higher bits should be provided by writing to the PCLATH register. It is illustrated in figure 2-17 below as a

jump to the subroutine PP2 address.

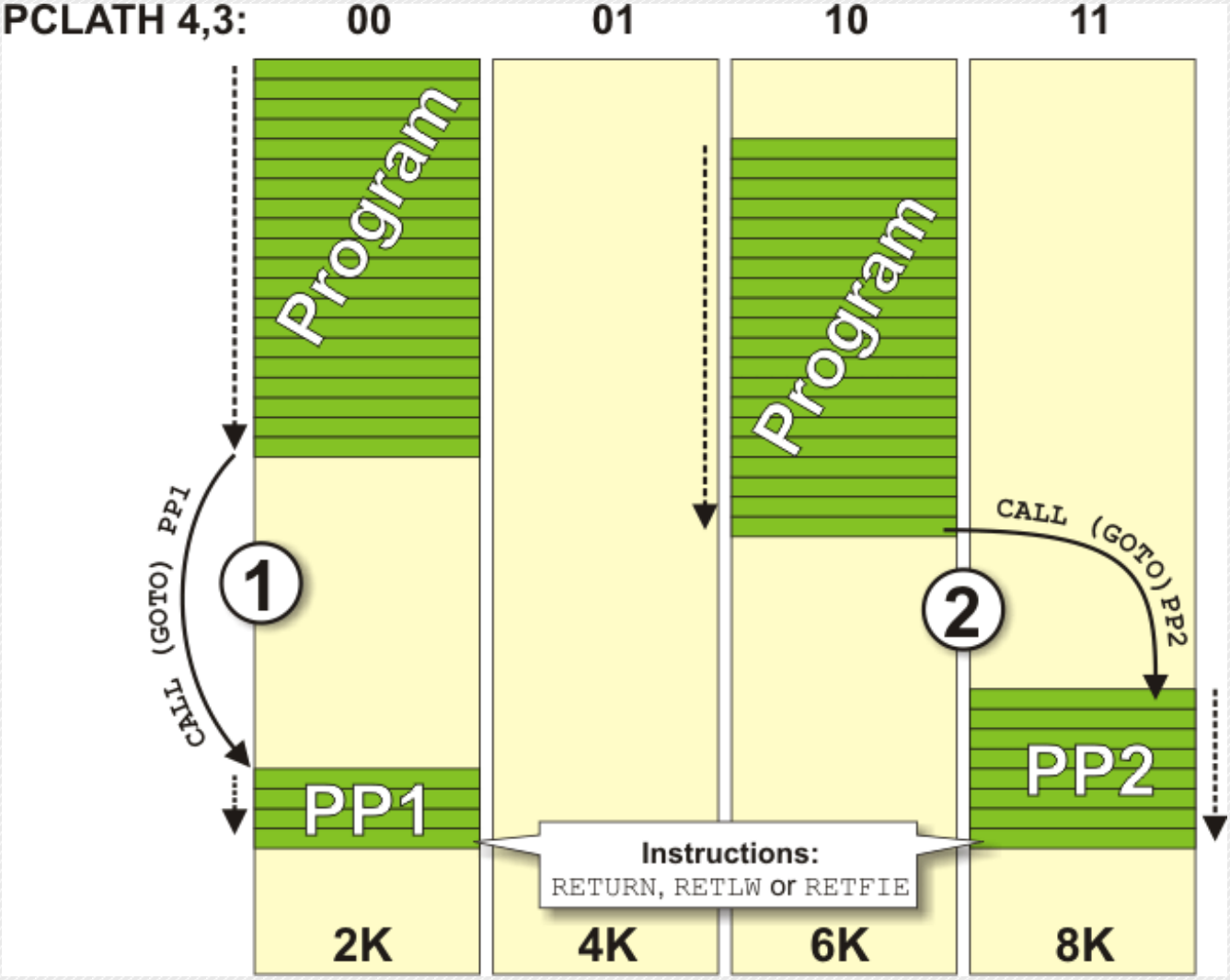


Fig. 2-17 PCLATH Registers

In both cases, when the subroutine reaches instructions `RETURN`, `RETLW` or `RETFIE` (to return to the main program), the microcontroller will simply continue program execution from where it left off because the return address is pushed and saved onto the stack which, as mentioned, consists of 13-bit registers.

Indirect addressing

In addition to direct addressing which is logical and clear by itself (it is sufficient to specify address of some register to read its contents), this microcontroller is able to perform indirect addressing by means of the INDF and FSR registers. It sometimes considerably simplifies program writing. The whole procedure is enabled because the INDF register is not true one (physically does not exist), but only specifies the register whose address is located in the FSR register. Because of this, write or read from the INDF register actually means write or read from the register whose address is located in the FSR register. In other words, registers' addresses are specified in the FSR register, and their contents are stored in the INDF register. The difference between direct and indirect addressing is illustrated in the figure 2-18 below:

As seen, the problem with the "missing addressing bits" is solved by "borrowing" from another register. This time, it is the seventh bit called IRP from the STATUS register.

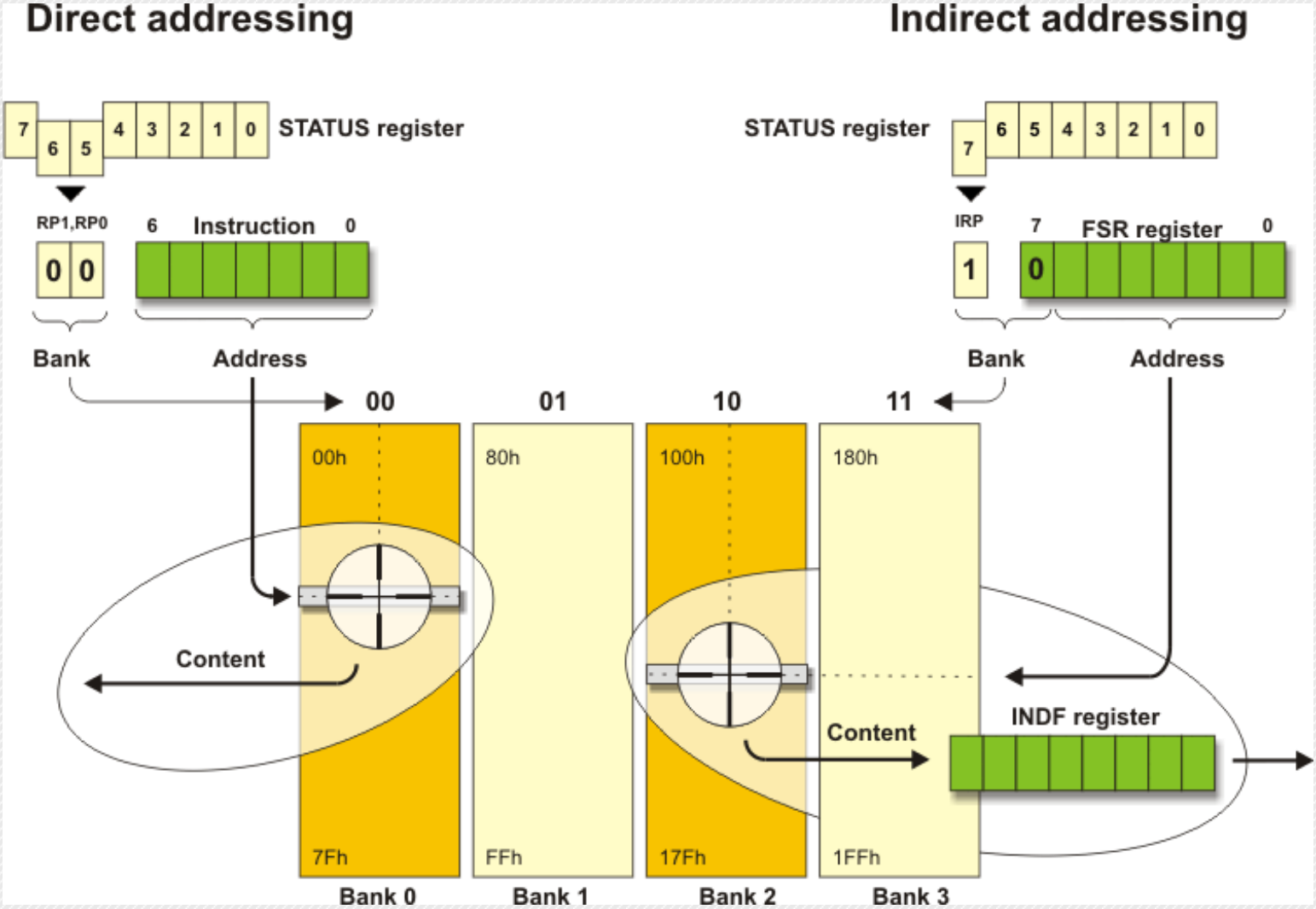


Fig. 2-18 Direct and Indirect addressing

Previous Chapter | Table of Contents | Next Chapter

Chapter 3: I/O Ports

Features and Functions

One of the most important feature of the microcontroller is a number of input/output pins used for connection with peripherals. In this case, there are in total of thirty-five general purpose I/O pins available, which is quite enough for the most applications.

In order pins' operation can match internal 8-bit organization, all of them are, similar to registers, grouped into five so called ports denoted by A, B, C, D and E. They all have several features in common:

- For practical reasons, many I/O pins have two or three functions. If a pin is used as any other function, it may not be used as a general purpose input/output pin; and
- Every port has its "satellite", i.e. the corresponding TRIS register: TRISA, TRISB, TRISC etc. which determines performance, but not the contents of the port bits.

By clearing some bit of the TRIS register (bit=0), the corresponding port pin is configured as output. Similarly, by setting some bit of the TRIS register (bit=1), the corresponding port pin is configured as input. This rule is easy to remember 0 = Output, 1 = Input.

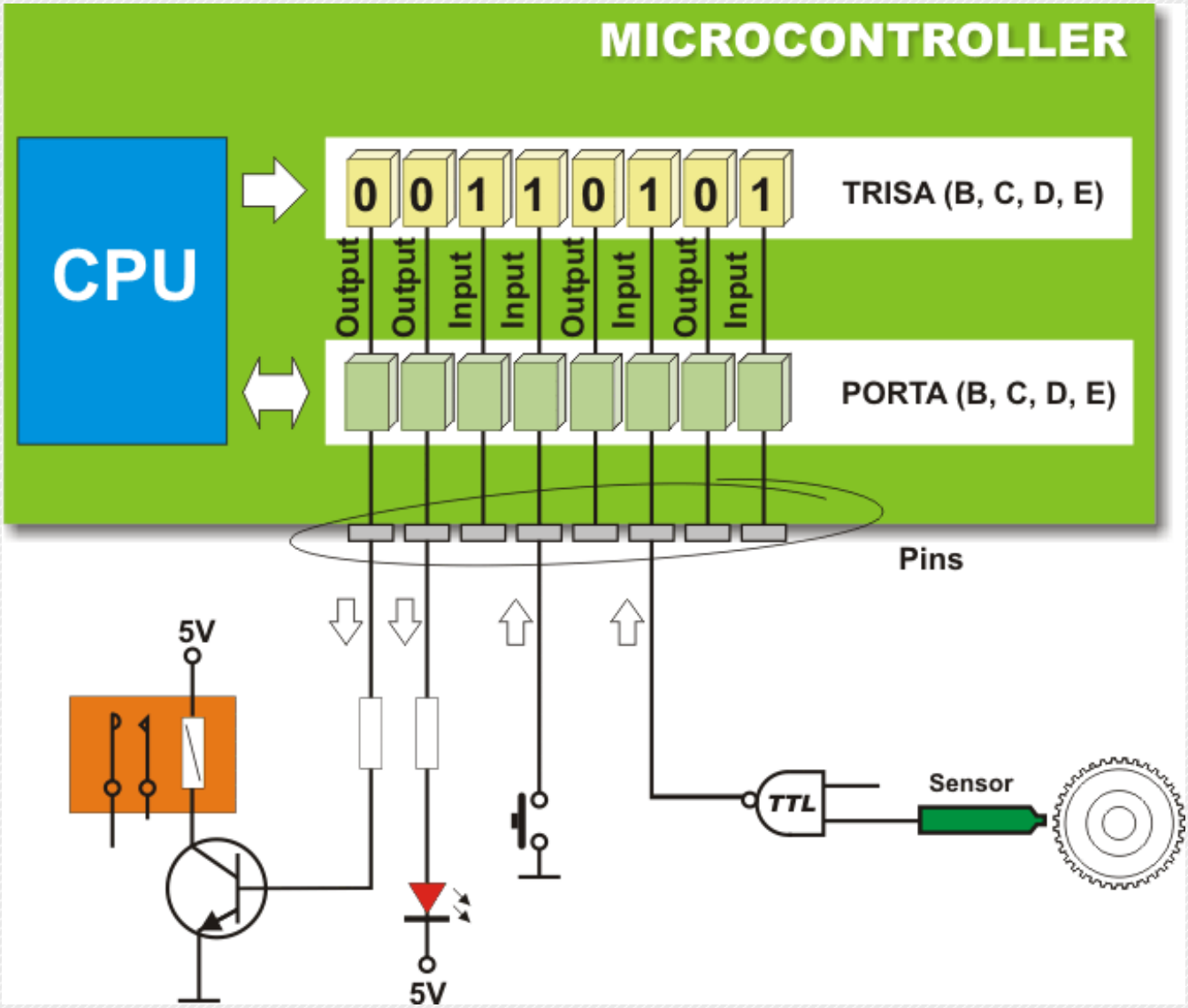


Fig. 3-1 I/O Ports

Port A and TRISA Register

Port A is an 8-bit wide, bidirectional port. Bits of the TRISA and ANSEL control the PORTA pins. All Port A pins act as digital inputs/outputs. Five of them can also be analog inputs (denoted as AN):

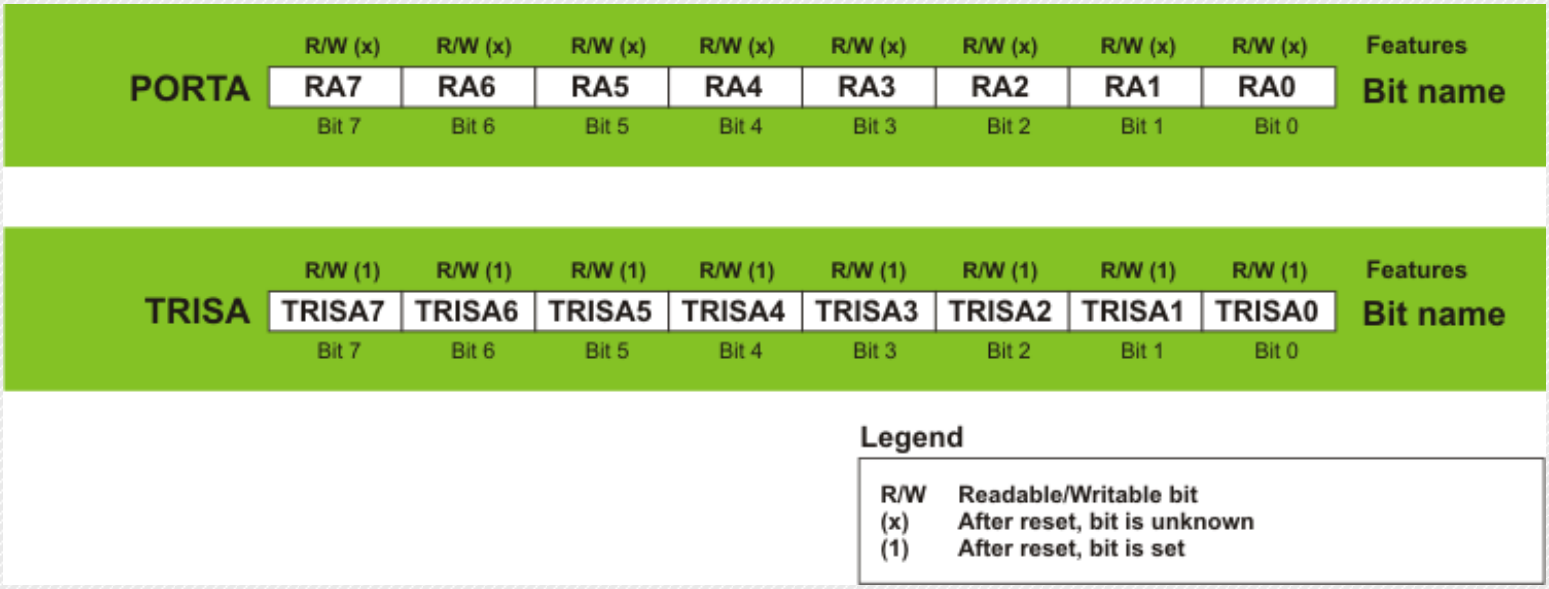


Fig. 3-2 Port A and TRISA Register

Similar to bits of the TRISA register which determine which of the pins will be configured as input and which as output, the appropriate bits of the ANSEL register determine whether the pins will act as analog inputs or digital inputs/outputs.

- RA0 = AN0 (determined by bit ANS0 of the ANSEL register);
- RA1 = AN1 (determined by bit ANS1 of the ANSEL register);
- RA2 = AN2 (determined by bit ANS2 of the ANSEL register);
- RA3 = AN3 (determined by bit ANS3 of the ANSEL register); and
- RA5 = AN4 (determined by bit ANS4 of the ANSEL register).

Each bit of this port has an additional function related to some of built-in peripheral units. These additional functions will be described in later chapters. This chapter covers only the RA0 pin's additional function since it is related to Port A only.

ULPWU Unit

The microcontroller is commonly used in devices which have to operate periodically and, completely independently using a battery power supply. In such cases, minimal power consumption is one of the priorities. Typical examples of such applications are: thermometers, sensors for fire detection and similar. It is known that a reduction in clock frequency reduces the power consumption, so one of the most convenient solutions to this problem is to slow the clock down (use 32KHz quartz crystal instead of 20MHz).

Setting the microcontroller to sleep mode is another step in the same direction. However, even when both measures are applied, another problem arises. How to wake the microcontroller and set it to normal mode. It is obviously necessary to have an external signal to change logic state on some of the pins. Thus, the problem still exists. This signal must be generated by additional electronics, which causes higher power consumption of the entire device.

The ideal solution would be the microcontroller wakes up periodically by itself, which is not impossible at all. The circuit which enables that is shown in figure on the right.

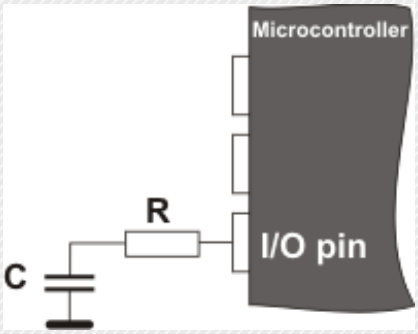


Fig. 3-3 ULPWU Unit

The principle of operation is simple:

A pin is configured as output and logic one (1) is brought to it. That causes the capacitor to be charged. Immediately after this, the same pin is configured as an input. The change of logic state enables an interrupt and the microcontroller is set to *Sleep* mode. Afterwards, there is nothing else to be done except wait for the capacitor to discharge by the leakage current flowing out through the input pin. When it occurs, an interrupt takes place and the microcontroller continues with the program execution in normal mode. The whole sequence is repeated...

Theoretically, this is a perfect solution. The problem is that all pins able to cause an interrupt in this way are digital and have relatively large leakage current when their voltage is not close to the limit values Vdd (1) or Vss (0). In this case, the capacitor is discharged for a short time since the current amounts to several hundreds of microamperes. This is why the ULPWU circuit able to register slow voltage drops with ultra low power consumption was designed. Its output generates an interrupt, while its input is connected to one of the microcontroller pins. It is the RA0 pin. Referring to Fig. 3-4 (R=200 ohms, C=1nF), discharge time is approximately 30mS, while the total consumption of the microcontroller is 1000 times lower (several hundreds of nanoamperes).

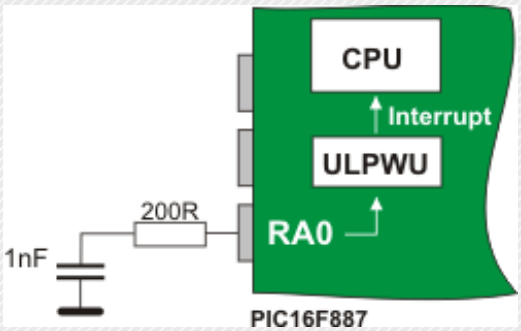


Fig. 3-4 Sleep Mode

Port B and TRISB Register

Port B is an 8-bit wide, bidirectional port. Bits of the TRISB register determine the function of its pins.

	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	R/W (x)	Features
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	Features
TRISB	TRISB7	TRISB6	TRISB5	TRISB4	TRISB3	TRISB2	TRISB1	TRISB0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

-	Bit is unimplemented
R/W	Readable/Writable bit
(x)	After reset, bit is unknown
(1)	After reset, bit is set

Fig. 3-5 Port B and TRISB register

Similar to Port A, a logic one (1) in the TRISB register configures the appropriate port pin as input and vice versa. Six pins on this port can act as analog inputs (AN). The bits of the ANSELH register determine whether these pins act as analog inputs or digital inputs/outputs:

- RB0 = AN12 (determined by bit ANS12 of the ANSELH register);
- RB1 = AN10 (determined by bit ANS10 of the ANSELH register);
- RB2 = AN8 (determined by bit ANS8 of the ANSELH register);
- RB3 = AN9 (determined by bit ANS9 of the ANSELH register);
- RB4 = AN11 (determined by bit ANS11 of the ANSELH register); and
- RB5 = AN13 (determined by bit ANS13 of the ANSELH register).

Each Port B pin has an additional function related to some of the built-in peripheral units, which will be explained in later chapters.

- All the port pins have built in *pull-up* resistor, which make them ideal for connection to push-buttons, switches and optocouplers. In order to connect these resistors to the microcontroller ports, the appropriate bit of the WPUB register should be set.*

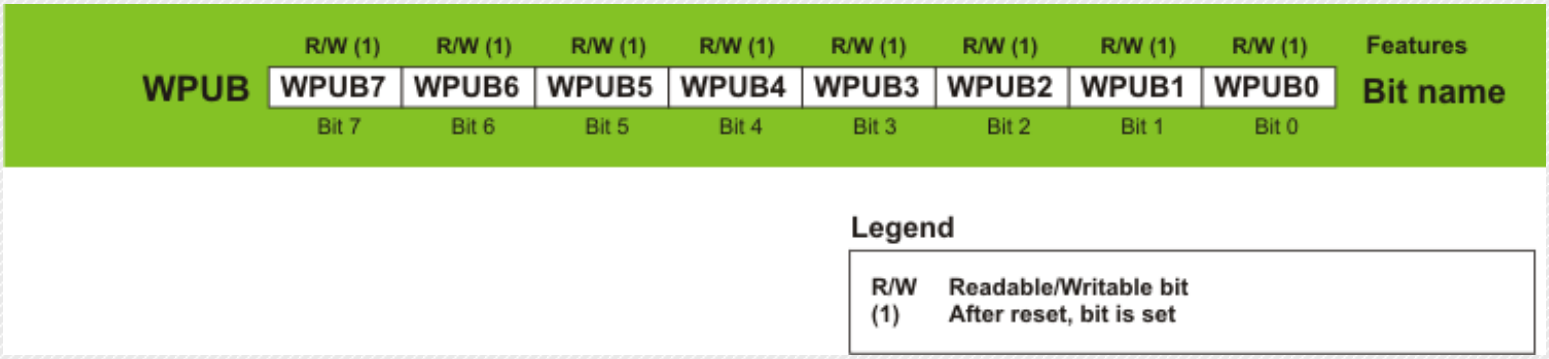


Fig. 3-6 WPUB register

Having a high level of resistance (several tens of kilo ohms), these “virtual” resistors do not affect pins configured as outputs, but serves as a useful complement to inputs. As such, they are connected to CMOS logic circuit inputs. Otherwise, they would act as if they are floating because of their high input resistance.

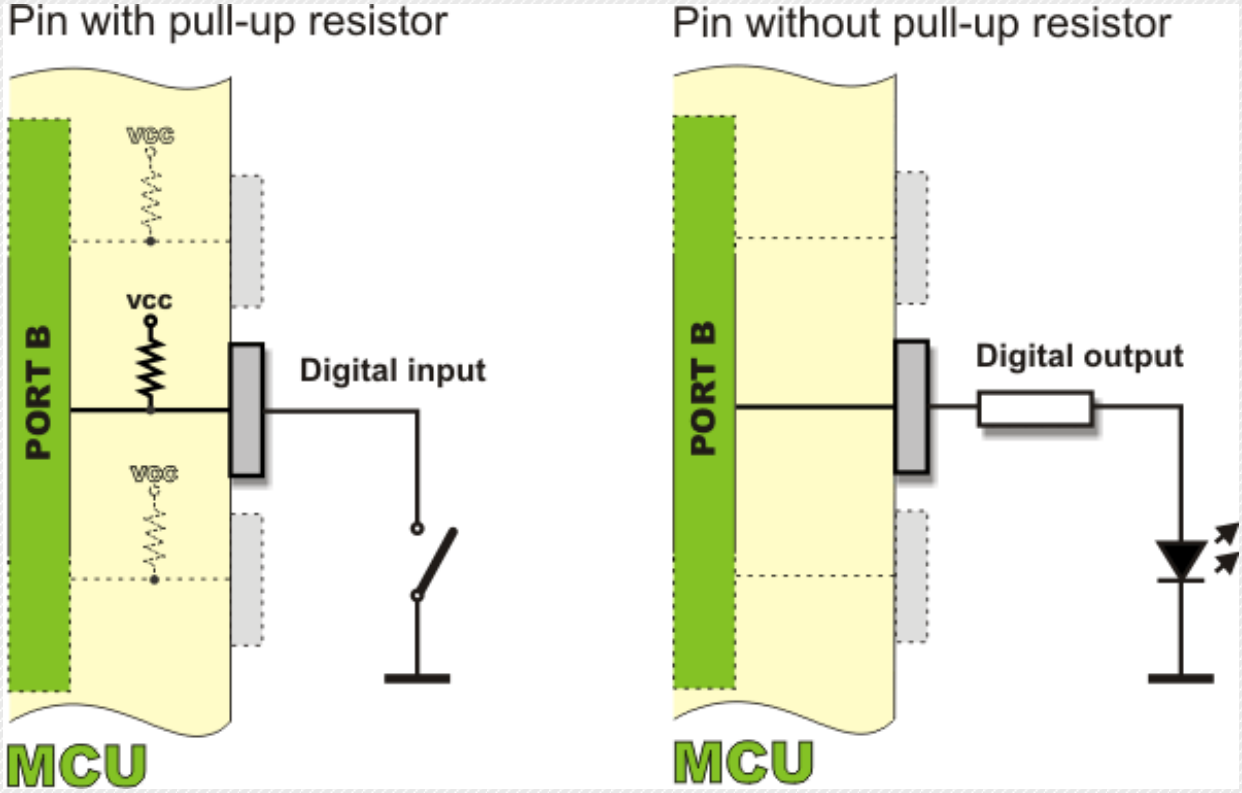


Fig. 3-7 Pull-up resistors

** Apart from the bits of the WPUB register, there is another bit affecting pull-up resistors installation. It is RBPU bit of the OPTION_REG. It is a general-purpose bit because it affects installation of all Port resistors.*

- If enabled, each Port B bit configured as an input may cause an interrupt by changeing its logic state. In order to enable pins to cause an interrupt, the appropriate bit of the IOCB register should be set.

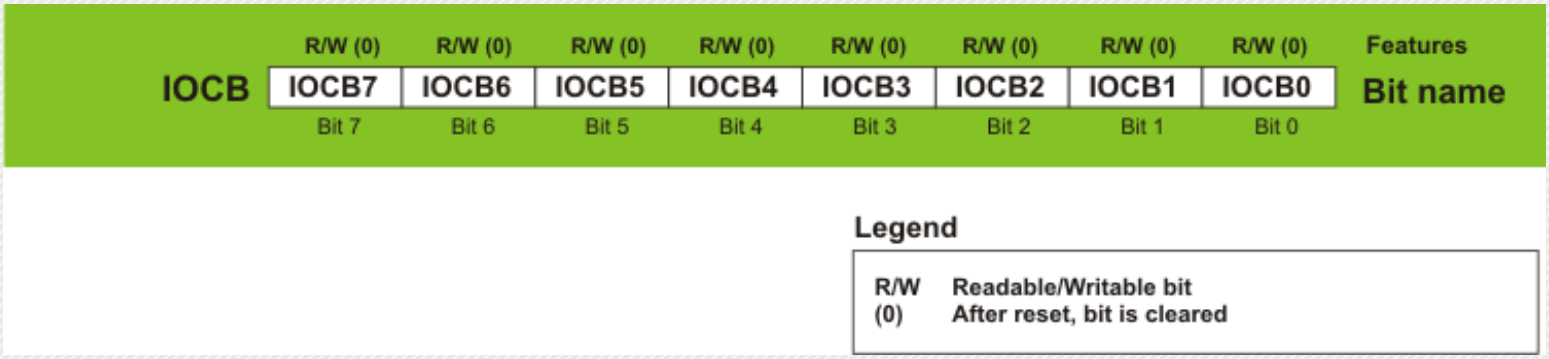


Fig. 3-8 IOCB register

Because of these features, the port B pins are commonly used for checking push-buttons on the keyboard because they unerringly register any button press. Therefore, there is no need to "scan" these inputs all the time.

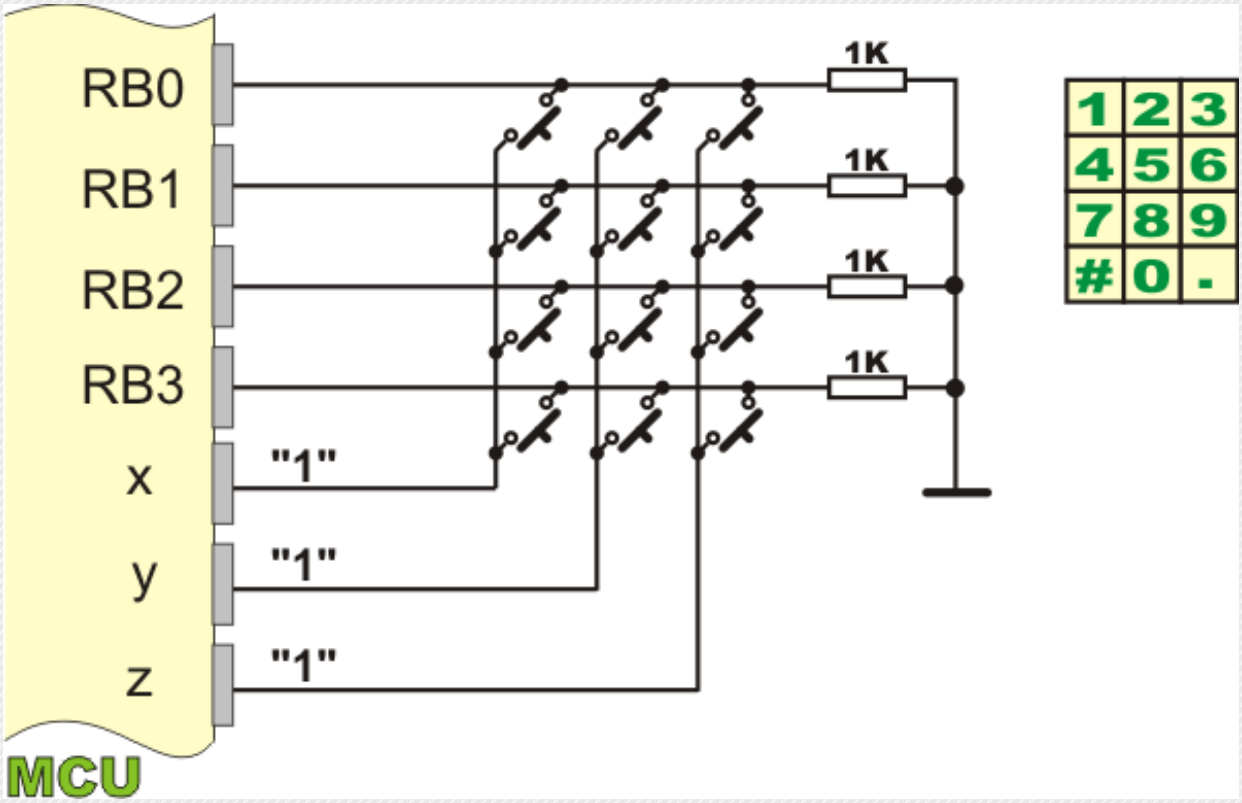


Fig. 3-9 Keyboard Example

When the X, Y and Z pins are configured as outputs set to logic one (1), it is only necessary to wait for an interrupt request which arrives upon any button press. By combining zeros and units on these outputs it is checked which push-button is pressed.

Pin RB0/INT

The RB0/INT pin is a single "true" external interrupt source. It can be configured to react to signal raising edge (zero-to-one transition) or signal falling edge (one-to-zero transition). The INTEDG bit of the OPTION_REG register selects the signal.

RB6 and RB7 Pins

You have probably noticed that the PIC16F887 microcontroller does not have any special pins for programming (writing the program to ROM). The Ports pins available as general purpose I/O pins during normal operation are used for this purpose (Port B pins clock (RB6) and data transfer (RB7) during program loading). In addition, it is necessary to apply a power supply voltage Vdd (5V) and Vss (0V), as well as voltage for FLASH memory programming Vpp (12-14V). During programming, Vpp voltage is applied to the MCLR pin. All details concerning this process, as well as which one of these

voltages is applied first, are beside the point, the programmers electronics are in charge of that. The point is that the program can be loaded to the microcontroller even when it is soldered onto the target device. Normally, the loaded program can also be changed in the same way. This function is called ICSP (In-Circuit Serial Programming). It is necessary to plan ahead when using it.

It is not complicated at all! It is only necessary to install a 4-pin connector onto the target device so that the necessary programmer voltages may be applied to the microcontroller. In order that these voltages don't interfere with other device electronics, design some sort of circuit breaking into this connection (using resistors or jumpers).

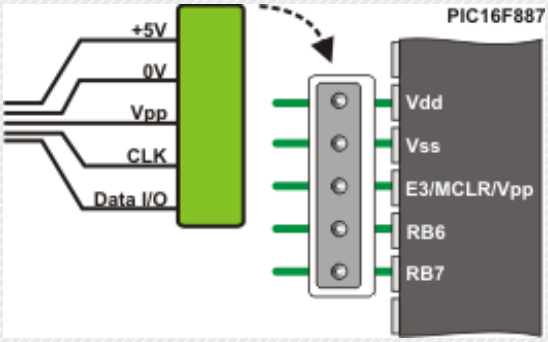
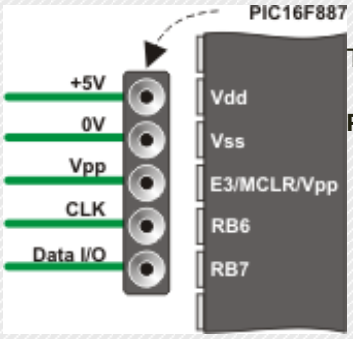


Fig. 3-10 ICSP Connection



These voltages are applied to socket pins in which the microcontroller is to be placed.

Fig. 3-11 Programmer On-Board Connections

Port C and TRISC Register

Port C is an 8-bit wide, bidirectional port. Bits of the TRISC Register determine the function of its pins. Similar to other ports, a logic one (1) in the TRISC Register configures the appropriate port pin as an input.

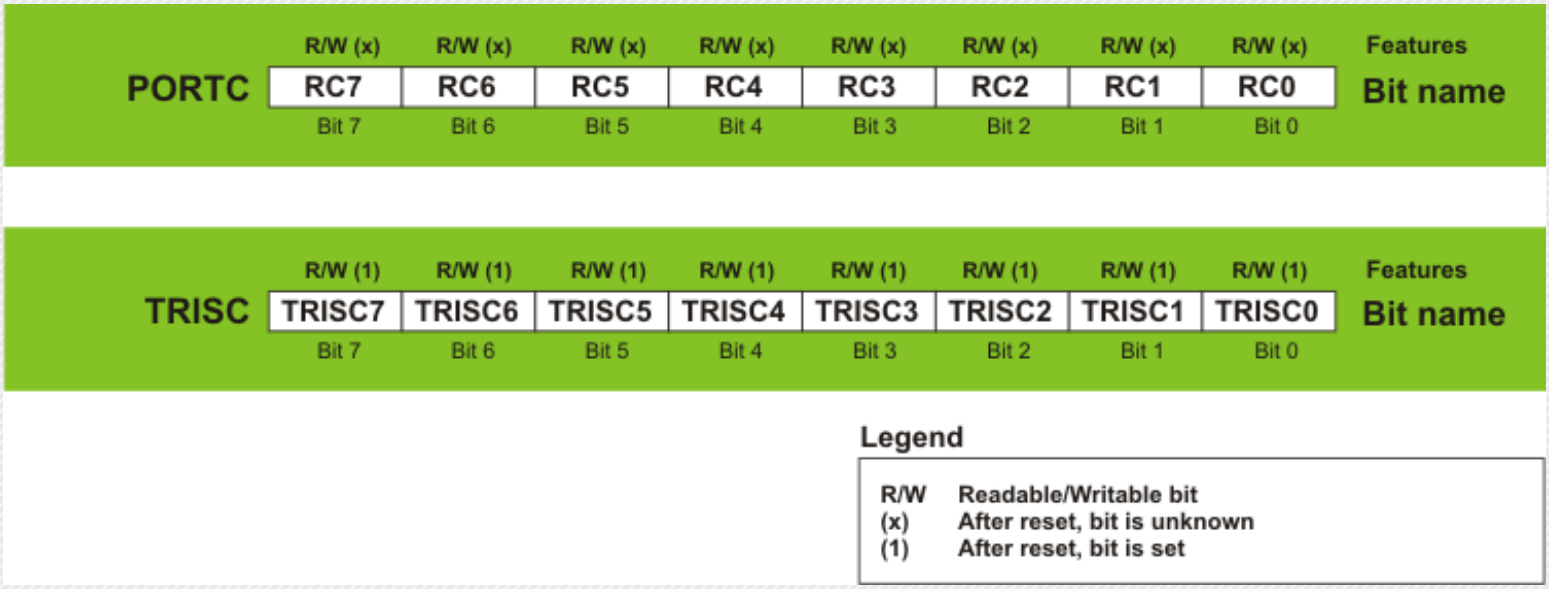


Fig. 3-12 Port C and TRISC Register

All additional functions of this port's bits will be explained later.

Port D and TRISD Register

Port D is an 8-bit wide, bidirectional port. Bits of the TRISD register determine the function of its pins. A logic one (1) in the TRISD register configures the appropriate port pin as input.

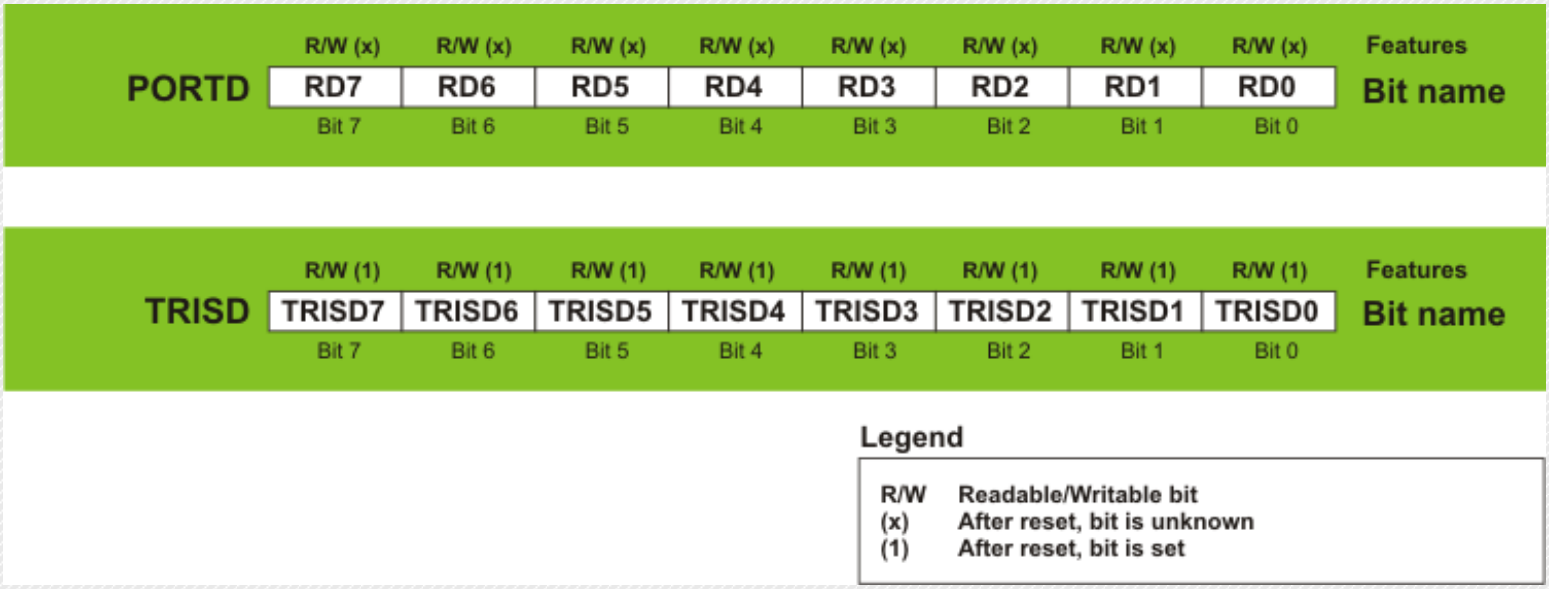


Fig. 3-13 Port D and TRISD Register

Port E and TRISE Register

Port E is a 4-bit wide, bidirectional port. The TRISE register's bits determine the function of its pins. Similar to other ports, a logic one (1) in the TRISE register configures the appropriate port pin as input. The exception is RE3 which is input only and its TRIS bit is always read as '1'.

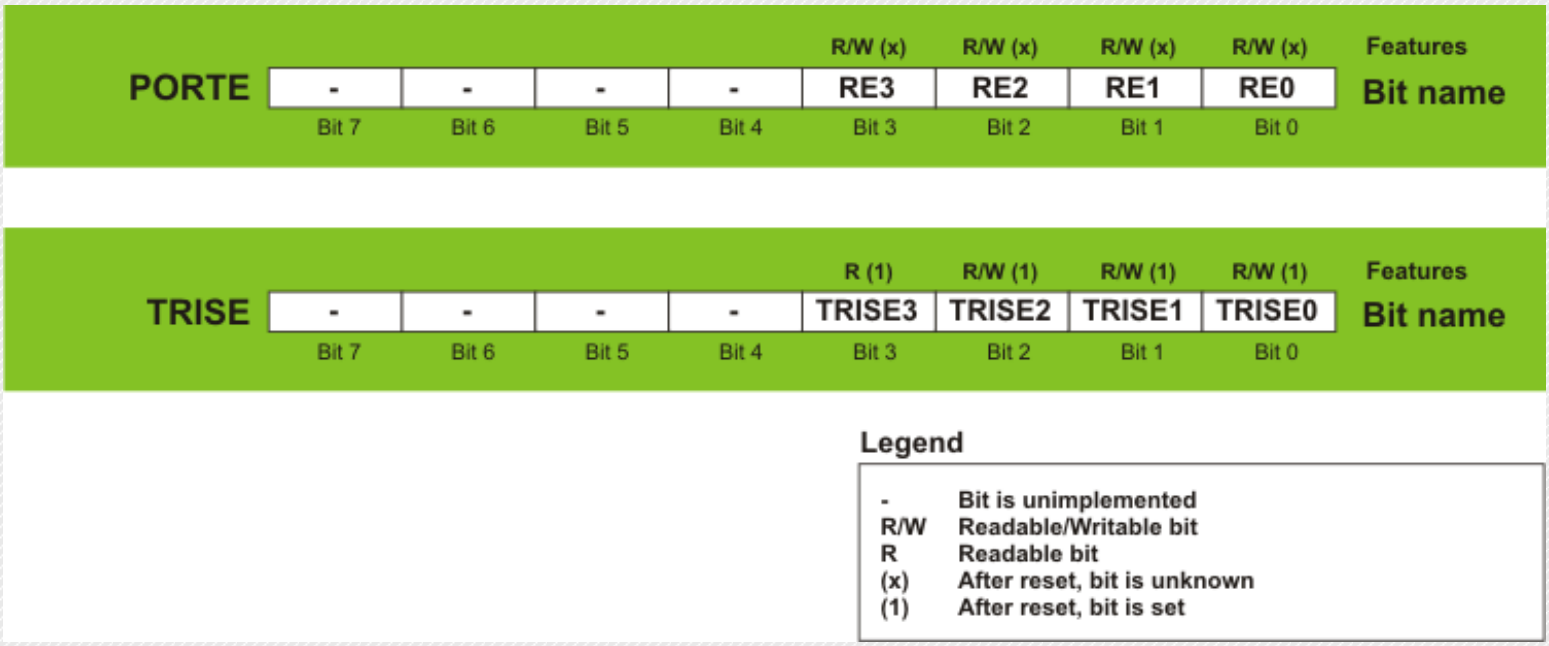


Fig. 3-14 Port E and TRISE Register

Similar to Ports A and B, three pins can be configured as analog inputs in this case. The ANSELH register bits determine whether a pin will act as analog input (AN) or digital input/output:

- RE0 = AN5 (determined by bit ANS5 of the ANSELregister);
- RE1 = AN6 (determined by bit ANS6 of the ANSELregister); and
- RE2 = AN7 (determined by bit ANS7 of the ANSELregister).

ANSEL and ANSELH Registers

The ANSEL and ANSELH registers are used to configure the input mode of an I/O pin to analog or digital.

ANSEL	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	Features
	ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

ANSELH			R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	Features
	-	-	ANS13	ANS12	ANS11	ANS10	ANS9	ANS8	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

-	Bit is unimplemented
R/W	Readable/Writable bit
(1)	After reset, bit is set

Fig. 3-15 ANSEL and ANSELH Registers

The rule is:
To configure a pin as an analog input, the appropriate bit of the ANSEL or ANSELH registers must be set (1). To configure pin as digital input/output, the appropriate bit must be cleared (0).

The state of the ANSEL bits has no affect on digital output functions. The result of any attempt to read some port pin configured as analog input will be 0.

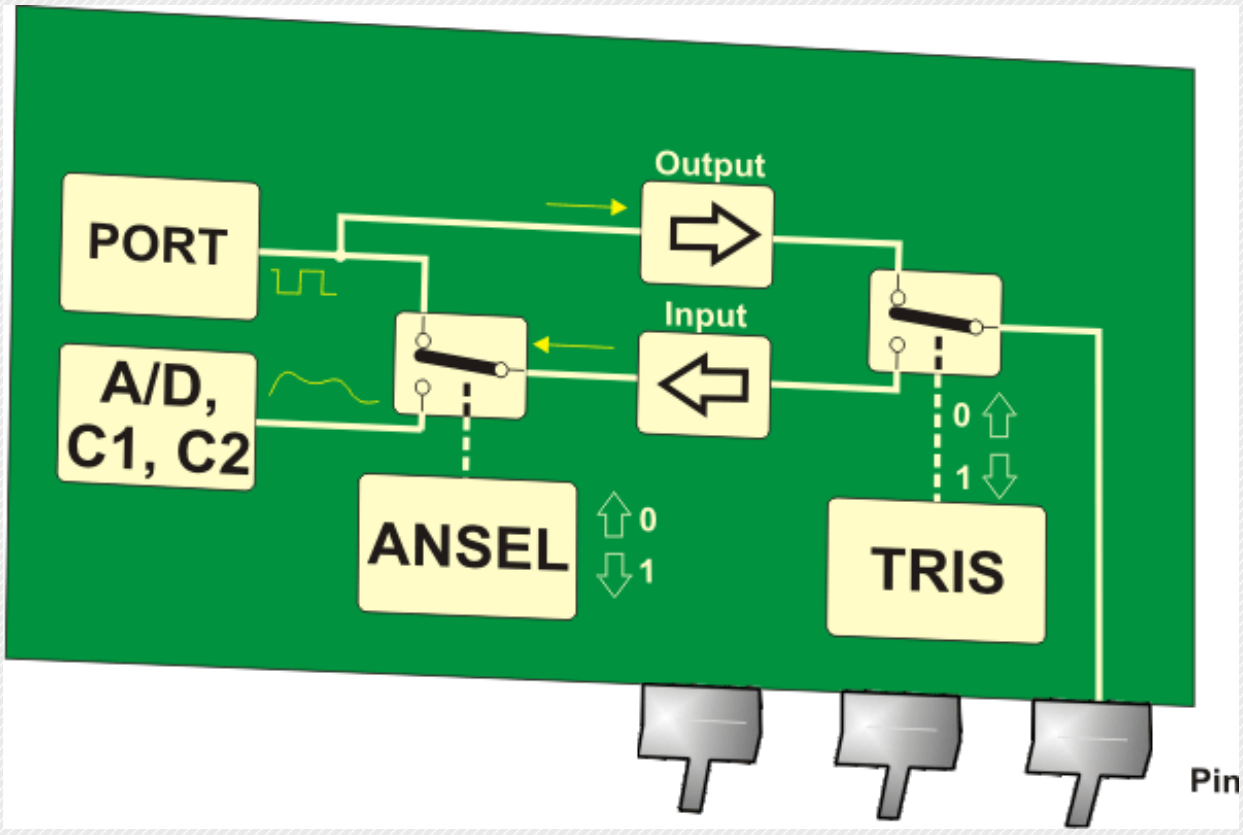


Fig. 3-16 ANSEL and ANSELH Configuration

In Short:

You will probably never write a program which fully utilises all the Ports in an efficient manner to justify learning all

there is to know about these Ports. However, they are probably the simplest modules within the microcontroller. This is how they are used:

- When designing a device, select a port through which the microcontroller will communicate to the peripheral environment. If you intend using only digital inputs/outputs, select any port you want. If you intend using some of the analog inputs, select the appropriate ports supporting such pins configuration (AN0-AN13);
- Each port pin may be configured as either input or output. Bits of the TRISA, TRISB, TRISC, TRISD and TRISE registers determine how the appropriate ports pins- PORTA, PORTB, PORTC, PORTD and PORTE will act;
- If you use some of the analog inputs, set the appropriate bits of the ANSEL and ANSELH registers at the beginning of the program;
- If you use switches and push-buttons as input signal source, connect them to Port B pins because they have pull-up resistors. The use of these resistors is enabled by the RBPU bit of the OPTION_REG register, whereas the installation of individual resistors is enabled by bits of the WPUB register; and
- It is usually necessary to react as soon as input pins change their logic state. How ever, it is not necessary to write a program for changing pins' logic state. It is far simpler to connect such inputs to the PORTB pins and enable the interrupt on every voltage change. Bits of the registers IOCOB and INTCON are in charge of that.

Previous Chapter | Table of Contents | Next Chapter

Chapter 4: Timers

The timers of the PIC16F887 microcontroller can be briefly described in only one sentence. There are three completely independent timers/counters marked as TMR0, TMR1 and TMR2. But it's not as simple as that.

Timer TMR0

The timer TMR0 has a wide range of applications in practice. Very few programs don't use it in some way. It is very convenient and easy to use for writing programs or subroutines for generating pulses of arbitrary duration, time measurement or counting external pulses (events) with almost no limitations.

The timer TMR0 module is an 8-bit timer/counter with the following features:

- 8-bit timer/counter;
- 8-bit prescaler (shared with Watchdog timer);
- Programmable internal or external clock source;
- Interrupt on overflow; and
- Programmable external clock edge selection.

Figure 4-1 below represents the timer TMR0 schematic with all bits which determine its operation. These bits are stored in the OPTION_REG Register.

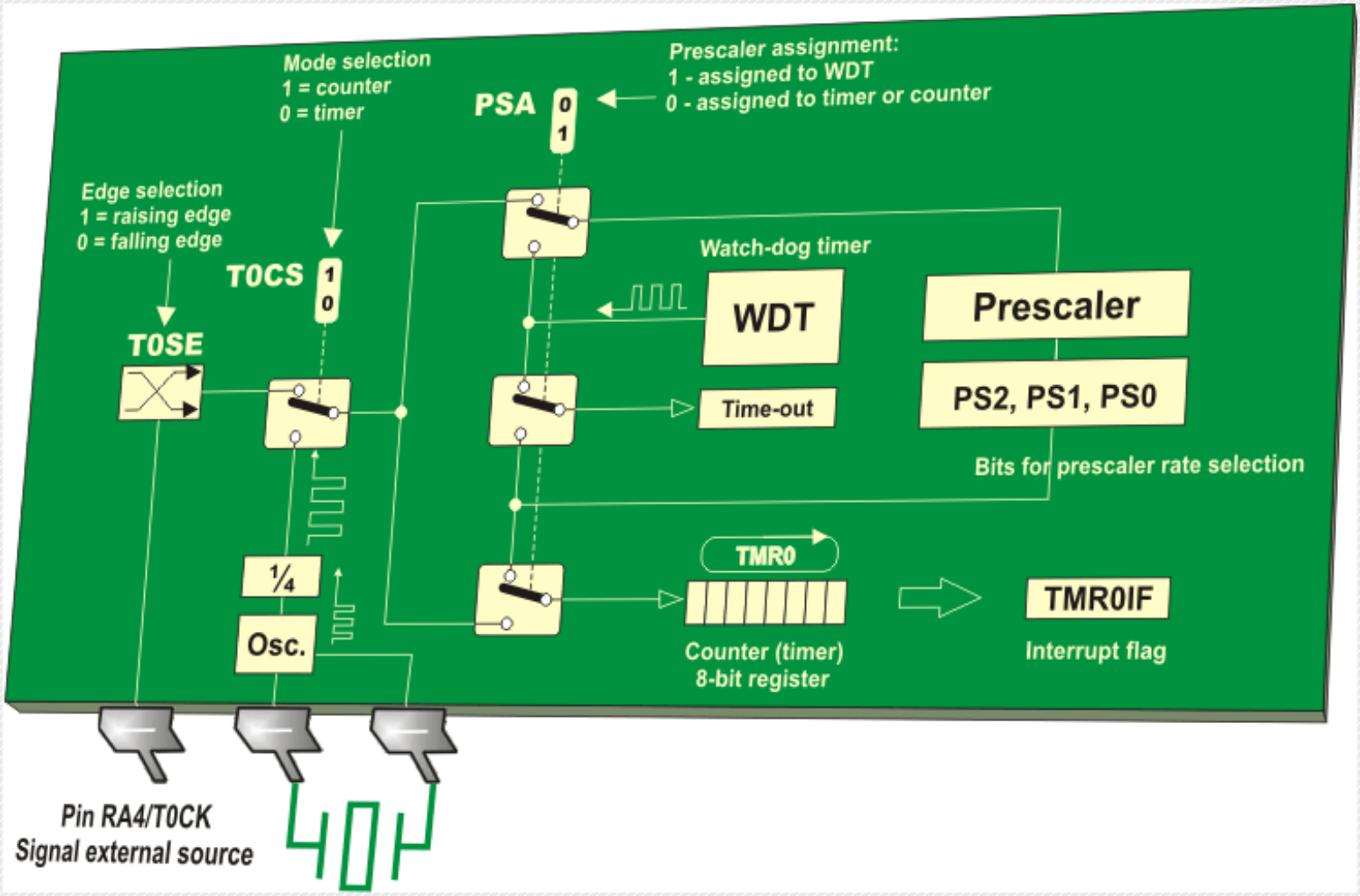


Fig. 4-1 Timer TMR0

OPTION_REG Register

	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	R/W (1)	Features
OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

R/W	Readable/Writable bit
(1)	After reset, bit is set

Fig. 4-2 OPTION_REG Register

- **RBPU - PORTB Pull-up enable bit**
  - 0 - PORTB pull-up resistors are disabled; and
  - 1 - PORTB pins can be connected to pull-up resistors.

- **INTEDG - Interrupt Edge Select bit**
  - 0 - Interrupt on rising edge of INT pin (0-1); and
  - 1 - Interrupt on falling edge of INT pin (1-0).

- **T0CS - TMR0 Clock Select bit**
  - 0 - Pulses are brought to TMR0 timer/counter input through the RA4 pin; and

- 1 - Internal cycle clock (Fosc/4).

- **T0SE - TMR0 Source Edge Select bit**
  - 0 - Increment on high-to-low transition on TMR0 pin; and
  - 1 - Increment on low-to-high transition on TMR0 pin.

- **PSA - Prescaler Assignment bit**
  - 0 - Prescaler is assigned to the WDT; and
  - 1 - Prescaler is assigned to the TMR0 timer/counter.

- **PS2, PS1, PS0 - Prescaler Rate Select bit**
  - Prescaler rate is adjusted by combining these bits
    As seen in the table 4-1, the same combination of bits gives different prescaler rate for the timer/counter and watch-dog timer respectively.

PS2	PS1	PS0	TMR0	WDT
0	0	0	1:2	1:1
0	0	1	1:4	1:2
0	1	0	1:8	1:4
0	1	1	1:16	1:8
1	0	0	1:32	1:16
1	0	1	1:64	1:32
1	1	0	1:128	1:64
1	1	1	1:256	1:128

Table 4-1 Prescaler Rate

The function of the PSA bit is shown in the two figures below:

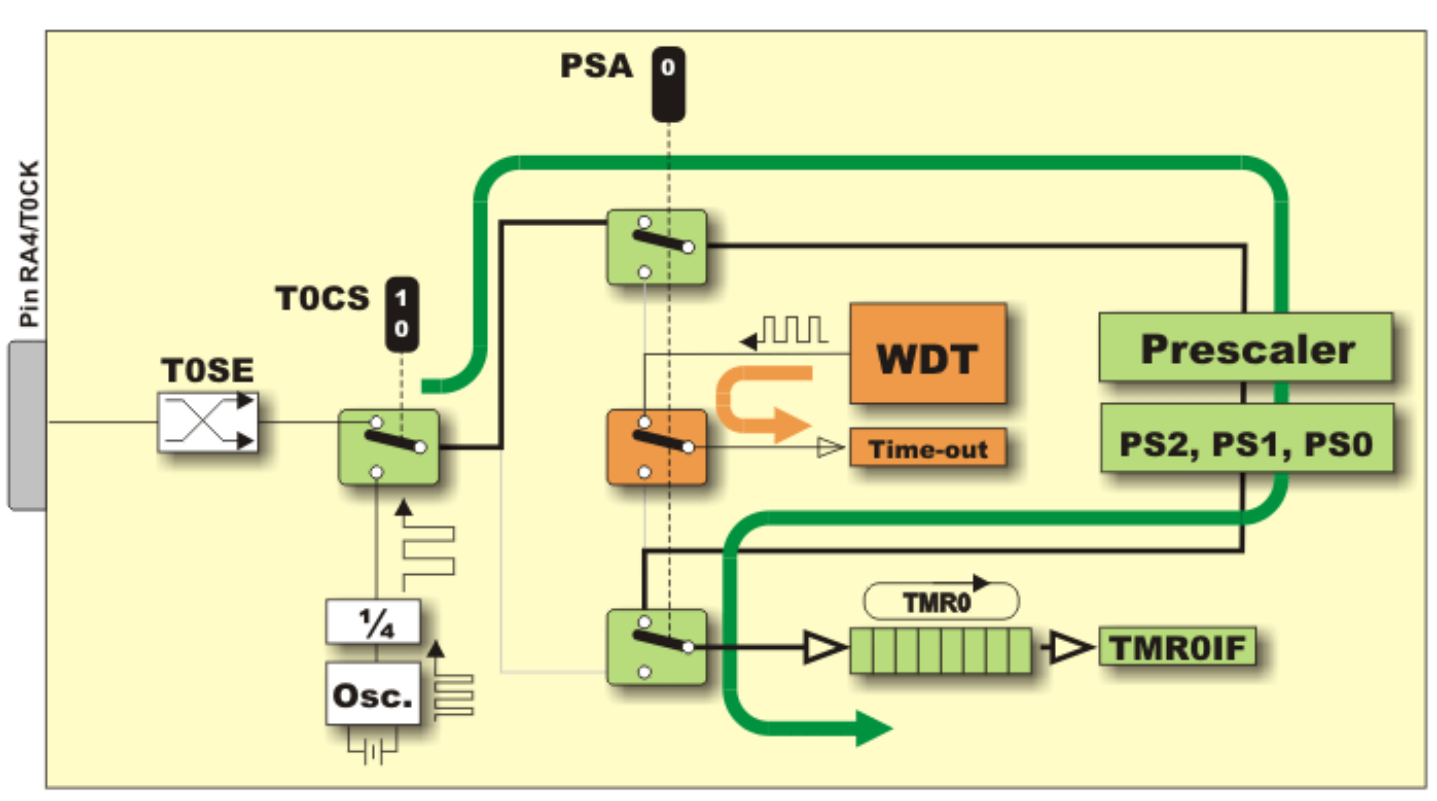


Fig. 4-3 The function of the PSA bit 0

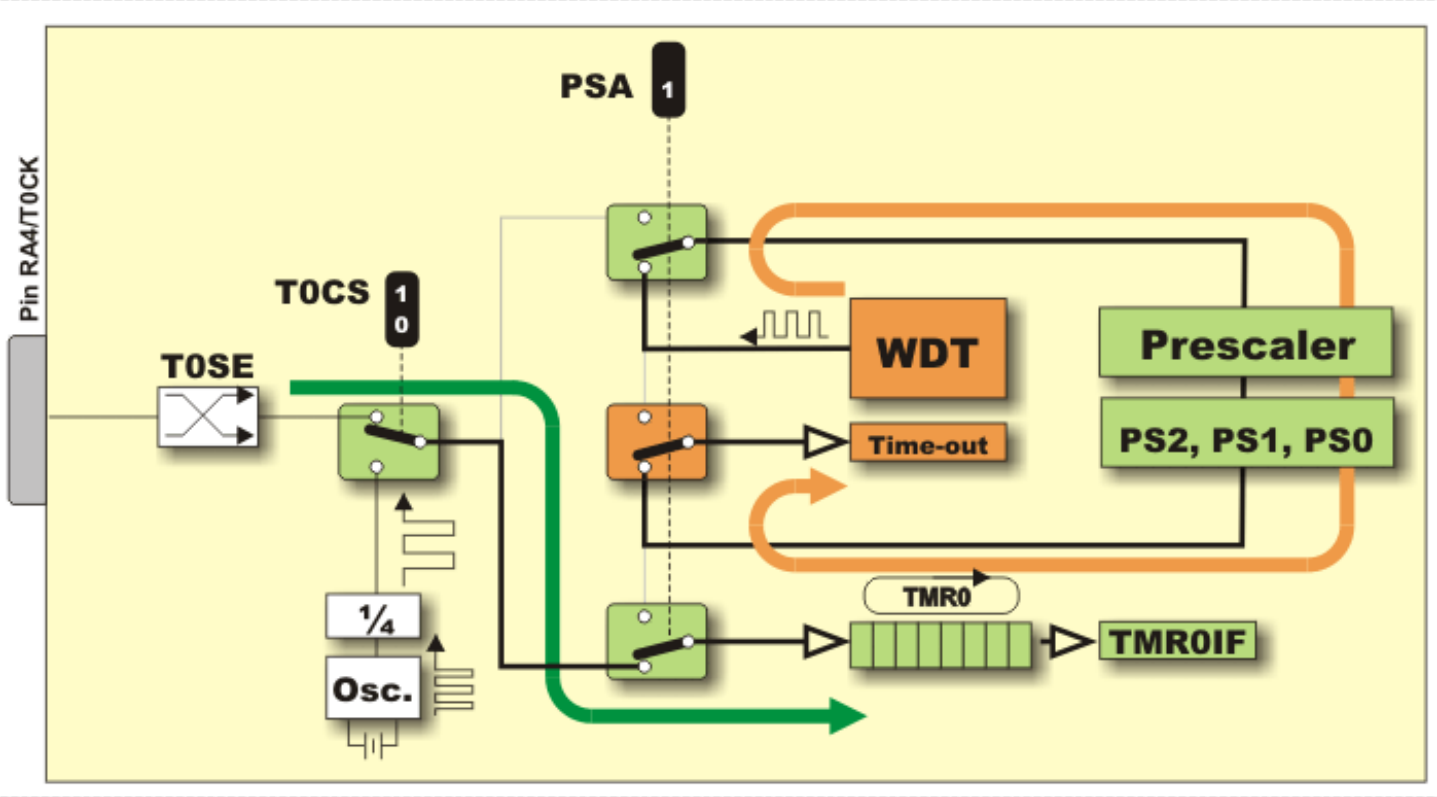


Fig. 4-4 The function of the PSA bit 1

As seen, the logic state of the PSA bit determines whether the prescaler is to be assigned to the timer/counter or watch-dog timer.

Additionally it is also worth mentioning:

- When the prescaler is assigned to the timer/counter, any write to the TMR0 register will clear the prescaler;
- When the prescaler is assigned to watch-dog timer, a CLRWDT instruction will clear both the prescaler and WDT;
- Writing to the TMR0 register used as a timer, will not cause the pulse counting to start immediately, but with two instruction cycles delay. Accordingly, it is necessary to adjust the value written to the TMR0 register;
- When the microcontroller is setup in *sleep* mode, the oscillator is turned off. Overflow cannot occur since there are no pulses to count. This is why the TMR0 overflow interrupt cannot wake up the processor from Sleep mode;
- When used as an external clock counter without prescaler, a minimal pulse length or a pause between two pulses must be 2 Tosc + 20 nS. Tosc is the oscillator signal period;
- When used as an external clock counter with prescaler, a minimal pulse length or a pause between two pulses is 10nS;
- The 8-bit prescaler register is not available to the user, which means that it cannot be directly read or written to;
- When changing the prescaler assignment from TMR0 to the watch-dog timer, the following instruction sequence must be executed in order to avoid reset:

```
BANKSEL TMR0
CLRWDT                  ;CLEAR WDT
CLRF    TMR0            ;CLEAR TMR0 AND PRESCALER
BANKSEL OPTION_REG
BSF     OPTION_REG,PSA  ;PRESCALER IS ASSIGNED TO THE WDT
CLRWDT                  ;CLEAR WDT
MOVLW   b'11111000'     ;SELECT BITS PS2,PS1,PS0 AND CLEAR
ANDWF   OPTION_REG,W    ;THEM BY INSTRUCTION "LOGICAL AND"
IORLW   b'00000101'     ;BITS PS2, PS1, AND PS0 SET
MOVWF   OPTION_REG      ;PRESCALER RATE TO 1:32
```

- Likewise, when changing the prescaler assignment from the WDT to the TMR0, the following instruction sequence must be executed:

```
BANKSEL TMR0
CLRWDT                  ;CLEAR WDT AND PRESCALER
BANKSEL OPTION_REG
MOVLW   b'11110000'     ;SELECT ONLY BITS PSA,PS2,PS1,PS0
ANDWF   OPTION_REG,W    ;CLEAR THEM AFTERWARDS BY INSTRUCTION
                        ;"LOGICAL AND"
IORLW   b'00000011'     ;PRESCALER RATE IS 1:16
MOVWF   OPTION_REG
```

In order to use TMR0 properly, it is necessary:

To select mode:

- Timer mode is selected by the T0CS bit of the OPTION_REG register, (T0CS: 0=timer, 1=counter);
- When used, the prescaler should be assigned to the timer/counter by clearing the PSA bit of the OPTION_REG register. The prescaler rate is set by using the PS2-PS0 bits of the same register; and
- When using interrupt, the GIE and TMR0IE bits of the INTCON register should be set.

To measure time:

- Reset the TMR0 register or write some well-known value to it;
- Elapsed time (in microseconds when using quartz 4MHz) is measured by reading the TMR0 register; and
- The flag bit TMR0IF of the INTCON register is automatically set every time the TMR0 register overflows. If enabled, an interrupt occurs.

To count pulses:

- The polarity of pulses are to be counted is selected on the RA4 pin are selected by the T0SE bit of the OPTION register (T0SE: 0=positive, 1=negative pulses); and
- Number of pulses may be read from the TMR0 register. The prescaler and interrupt are used in the same manner as in timer mode.

Timer TMR1

Timer TMR1 module is a 16-bit timer/counter, which means that it consists of two registers (TMR1L and TMR1H). It can count up 65.535 pulses in a single cycle, i.e. before the counting starts from zero.

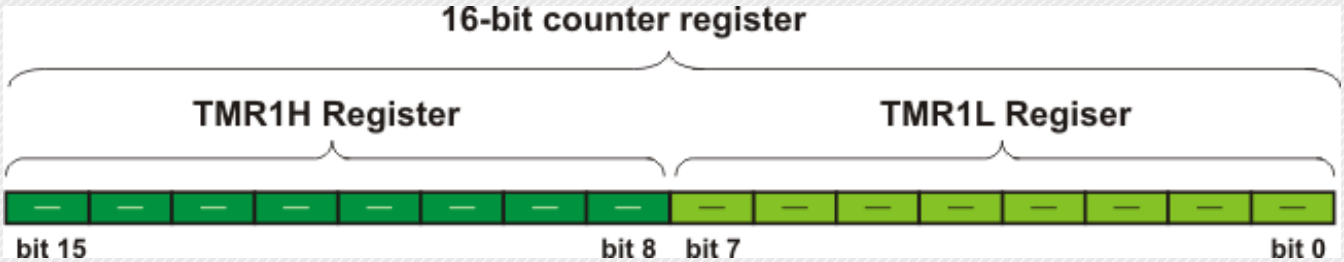


Fig. 4-5 Timer TMR1

Similar to the timer TMR0, these registers can be read or written to at any moment. In case an overflow occurs, an interrupt is generated.

The timer TMR1 module may operate in one of two basic modes- as a timer or a counter. However, unlike the timer TMR0, each of these modules has additional functions.

Parts of the T1CON register are in control of the operation of the timer TMR1.

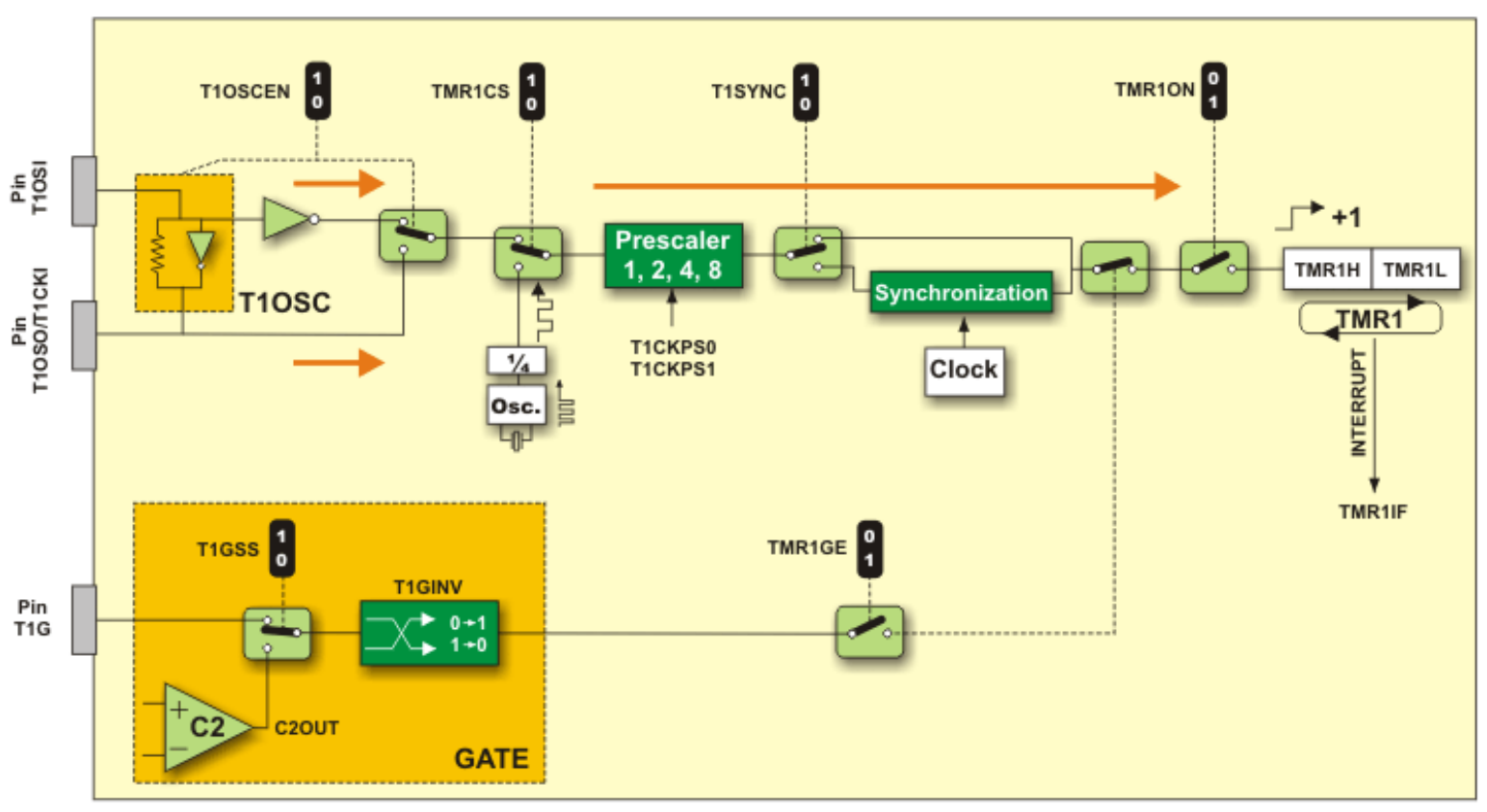


Fig. 4-6 Timer TMR1 Overview

Timer TMR1 Prescaler

Timer TMR1 has a completely separate prescaler which allows 1, 2, 4 or 8 divisions of the clock input. The prescaler is not directly readable or writable. However, the prescaler counter is automatically cleared upon write to the TMR1H or TMR1L register.

Timer TMR1 Oscillator

RC0/T1OSO and RC1/T1OSI pins are used to register pulses coming from peripheral electronics, but they also have an additional function. As seen in figure 4-7, they are simultaneously configured as both input (pin RC1) and output (pin RC0) of the additional LP quartz oscillator (low power).

This additional circuit is primarily designed for operating at low frequencies (up to 200 KHz), more precisely, for using the 32,768 KHz quartz crystal. Such crystals are used in quartz watches because it is easy to obtain one-second-long pulses by simply dividing this frequency.

Since this oscillator does not depend on internal clocking, it can operate even in *sleep* mode. It is enabled by setting the T1OSCEN control bit of the T1CON register. The user must provide a software time delay (a few milliseconds) to ensure proper oscillator start-up.

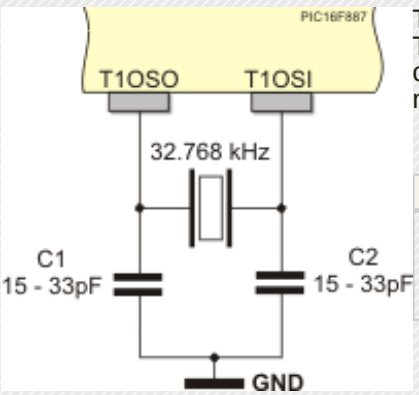


Table below shows the recommended values of capacitors to suit the quartz oscillator. These values do not have to be exact. However, the general rule is: the higher the capacitor's capacity the higher the stability, which, at the same time, prolongs the time needed for the oscillator stability.

Oscillator	Frequency	C1	C2
LP	32 kHz	33 pF	33 pF
	100 kHz	15 pF	15 pF
	200 kHz	15 pF	15 pF

Fig. 4-7 Timer TMR1 Oscillator

Timer TMR1 Gate

Timer 1 gate source is software configurable to be the T1G pin or the output of comparator C2. This gate allows the timer to directly time external events using the logic state on the T1G pin or analog events using the comparator C2 output. Refer to figure 4-7 above. In order to time a signals duration it is sufficient to enable such gate and count pulses having passed through it.

TMR1 in timer mode

In order to select this mode, it is necessary to clear the TMR1CS bit. After this, the 16-bit register will be incremented on every pulse coming from the internal oscillator. If the 4MHz quartz crystal is in use, it will be incremented every microsecond.

In this mode, the T1SYNC bit does not affect the timer because it counts internal clock pulses. Since the whole electronics uses these pulses, there is no need for synchronization.

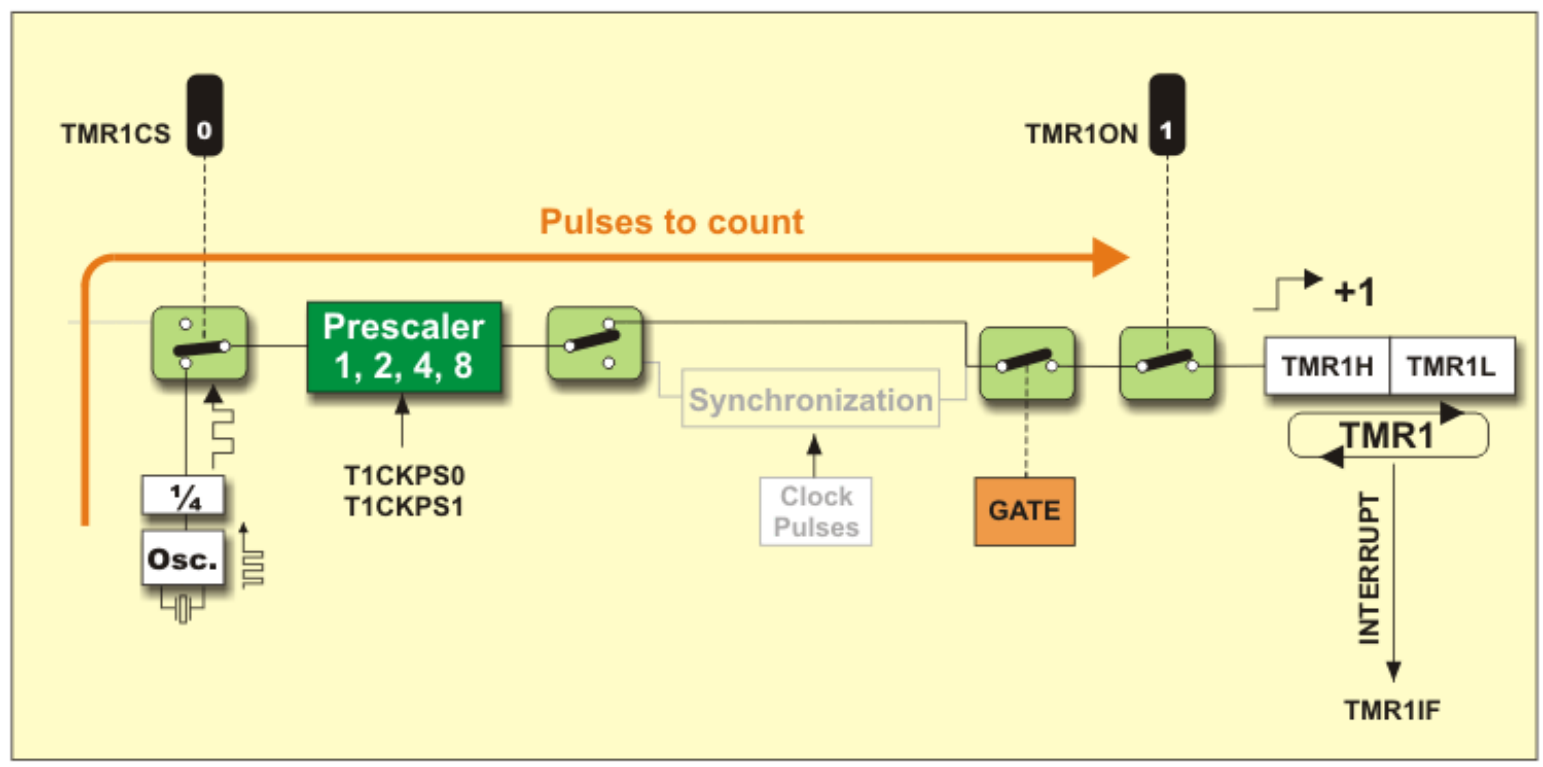


Fig. 4-8 TMR1 in timer mode

The microcontroller's clock oscillator does not run during sleep mode so the timer register overflow cannot cause any interrupt.

Timer TMR1 Oscillator

The power consumption of the microcontroller is reduced to the lowest level in *Sleep* mode. The point is to stop the oscillator. Anyway, it is easy to set the timer in this mode- by writing a SLEEP instruction to the program. A problem occurs when it is necessary to wake up the microcontroller because only an interrupt can do that. Since the microcontroller "sleeps", an interrupt must be triggered by external electronics. It can all get incredibly complicated if it is necessary the 'wake up' occurs at regular time intervals...

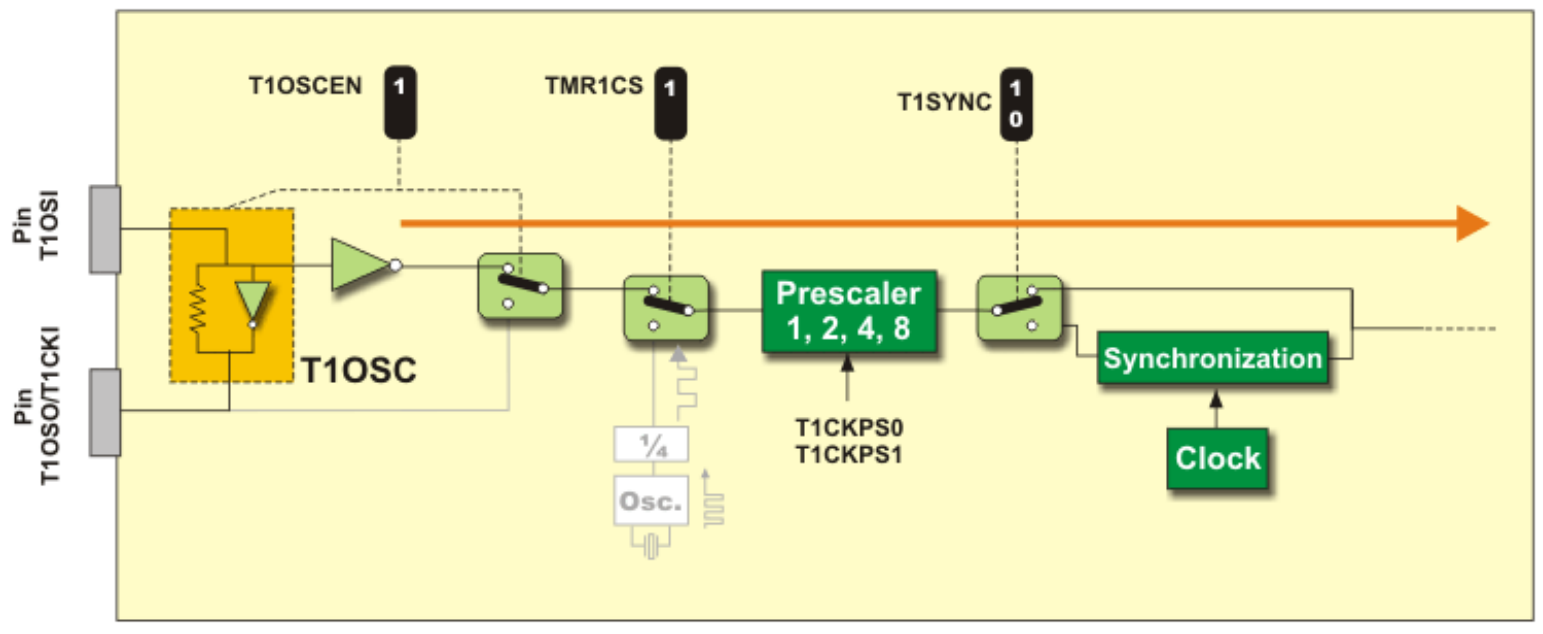


Fig. 4-9 Timer TMR1 Oscillator

In order to solve this problem, a completely independent *Low Power* quartz oscillator, able to operate in *sleep mode*, is built into the PIC16F887 microcontroller. Simply, what previously has been a separate circuit, it is now built into the microcontroller and assigned to the timer TMR1. The oscillator is enabled by setting the T1OSCEN bit of the T1CON register. After that, the TMR1CS bit of the same register then is used to determine that the timer TMR1 uses pulse sequences from that oscillator.

- The signal from this quartz oscillator is synchronized with the microcontroller clock by clearing the T1SYNC bit. In that case, the timer cannot operate in *sleep* mode. You wonder why? Because the circuit for synchronization uses the clock of microcontroller!; and
- The TMR1 register overflow interrupt may be enabled. Such interrupts will occur in *sleep* mode as well.

TMR1 in counter mode

Timer TMR1 starts to operate as a counter by setting the TMR1CS bit. It means that the timer TMR1 is incremented on the rising edge of the external clock input T1CKI. If control bit T1SYNC of the T1CON register is cleared, the external clock inputs will be synchronized on their way to the TMR1 register. In other words, the timer TMR1 is synchronized to the microcontroller system clock and called a synchronous counter.

When the microcontroller ,operating in this way, is set in *sleep* mode, the TMR1H and TMR1L timer registers are not incremented even though clock pulses appear on the input pins. Simply, since the microcontroller system clock does not run in this mode, there are no clock inputs to use for synchronization. However, the prescaler will continue to run if there are clock pulses on the pins since it is just a simple frequency divider.

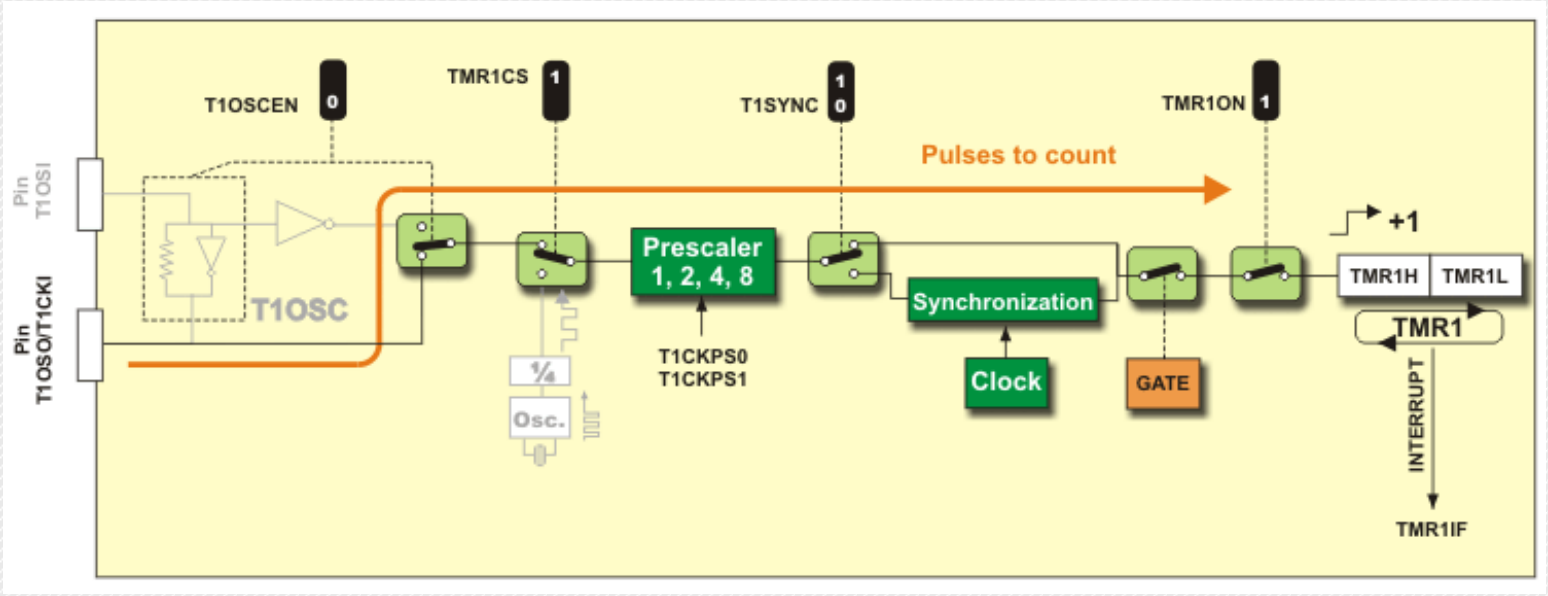
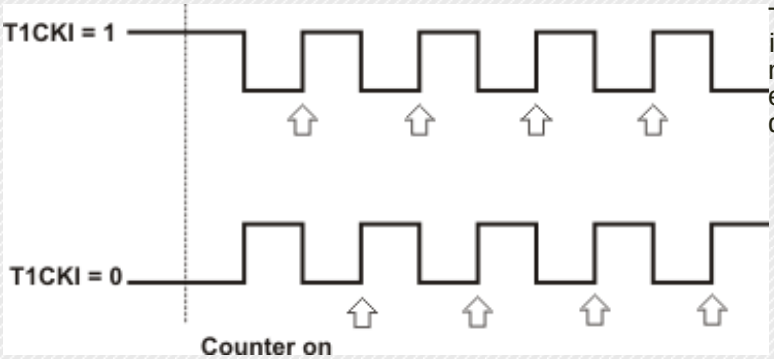


Fig. 4-11 Counter Mode



This counter registers a logic one (1) on input pins. It is important to understand that at least one falling edge must be registered prior to the first increment on rising edge. Refer to figure on the left. The arrows in figure 4-11 denote counter increments.

T1CON Register

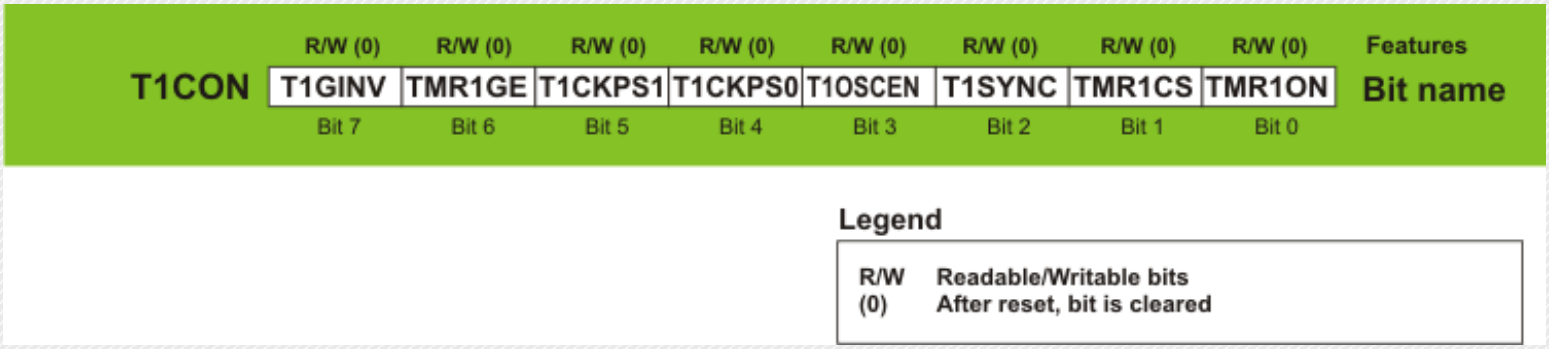


Fig. 4-12 T1CON Register

T1GINV - Timer1 Gate Invert bit acts as logic state inverter on the T1G pin gate or the comparator C2 output (C2OUT) gate. It enables the timer to mea sure time whilst the gate is high or low.

- 1 - Timer 1 counts when the pin T1G or bit C2OUT gate is high (1); and
- 0 - Timer 1 counts when the pin T1G or bit C2OUT gate is low (0).

TMR1GE - Timer1 Gate Enable bit determines whether the pin T1G or comparator C2 output (C2OUT) gate will be active or not. This bit is functional only in the event that the timer TMR1 is on (bit TMR1ON = 1). Otherwise, this bit is ignored.

- 1 Timer TMR1 is on only if timer 1 gate is not active; and
- 0 Gate does not affect the timer TMR1.

T1CKPS1, T1CKPS0 - Timer1 Input Clock Prescale Select bits determine the rate of the prescaler assigned to the timer TMR1.

T1CKPS1	T1CKPS0	Prescaler Rate
0	0	1:1
0	1	1:2
1	0	1:4
1	1	1:8

Table 4-2 Prescaler Rate

T1OSCEN - LP Oscillator Enable Control bit

- 1 - LP oscillator is enabled for timer TMR1 clock (oscillator with low power consumption and frequency 32.768 kHz); and
- 0 - LP oscillator is off.

T1SYNC - Timer1 External Clock Input Synchronization Control bit enables synchronization of the LP oscillator input or T1CKI pin input with the microcontroller internal clock. When counting pulses from the local clock source (bit TMR1CS = 0), this bit is ignored.

- 1 - Do not synchronize external clock input; and
- 0 - Synchronize external clock input.

TMR1CS - Timer TMR1 Clock Source Select bit

- 1 - Counts pulses on the T1CKI pin (on the rising edge 0-1); and
- 0 - Counts pulses of the internal clock of microcontroller.

TMR1ON - Timer1 On bit

- 1 - Enables Timer TMR1; and
- 0 - Stops Timer TMR1.

In order to use the timer TMR1 properly, it is necessary to perform the following:

- Since it is not possible to turn off the prescaler, its rate should be adjusted by using bits T1CKPS1 and T1CKPS0 of the register T1CON (Refer to table 4-2);
- The mode should be selected by the TMR1CS bit of the same register (TMR1CS: 0= the clock source is quartz oscillator, 1= the clock source is supplied externally);
- By setting the T1OSCEN bit of the same register, the timer TMR1 is turned on and the TMR1H and TMR1L registers are incremented on every clock input. Counting stops by clearing this bit;
- The prescaler is cleared by clearing or writing the counter registers; and
- By filling both timer registers, the flag TMR1IF is set and counting starts from zero.

Timer TMR2

Timer TMR2 module is an 8-bit timer which operates in a very specific way.

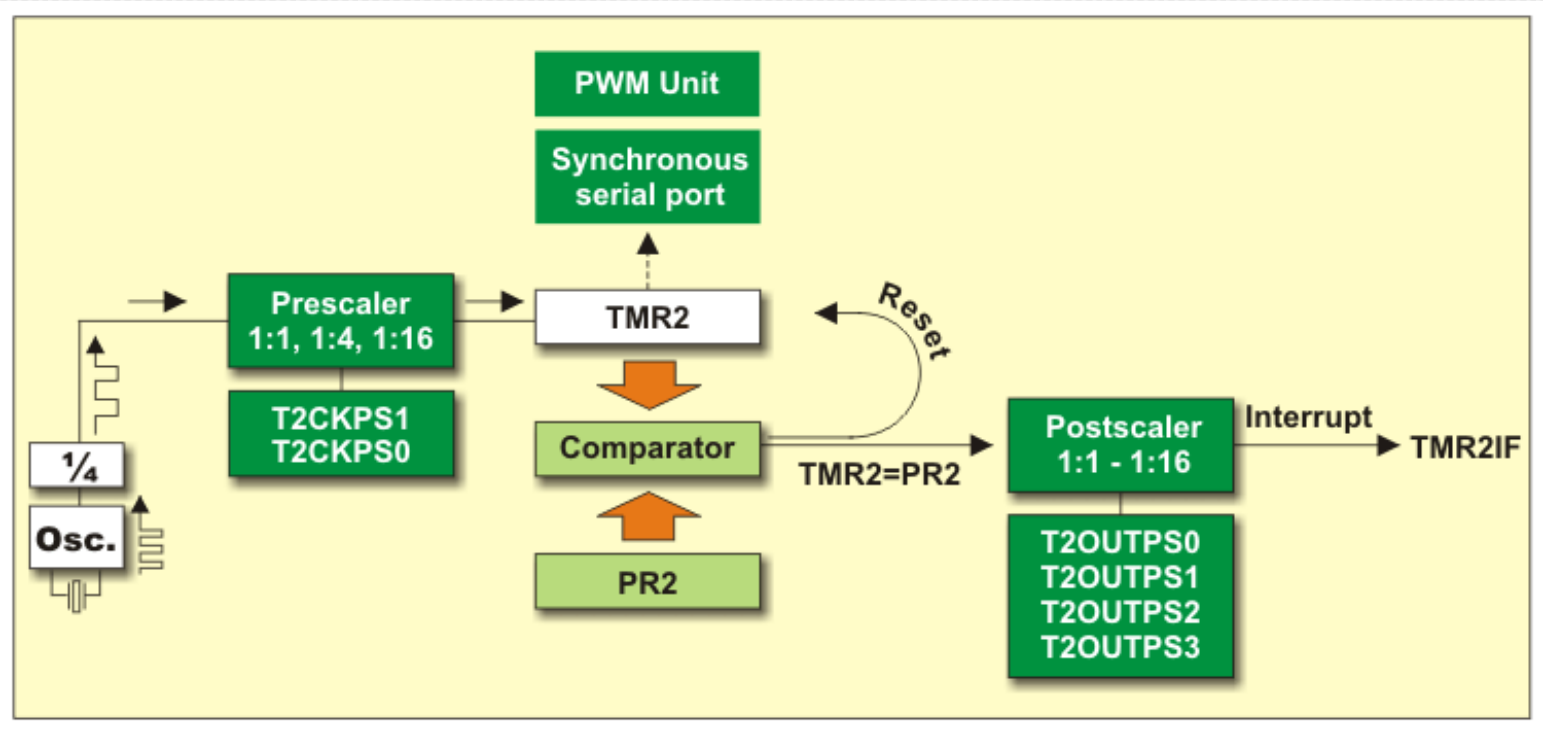


Fig. 4-13 Timer TMR2

The pulses from the quartz oscillator first pass through the prescaler whose rate may be changed by combining the T2CKPS1 and T2CKPS0 bits. The output of the prescaler is then used to increment the TMR2 register starting from 00h. The values of TMR2 and PR2 are constantly compared and the TMR2 register keeps on being incremented until it matches the value in PR2. When a match occurs, the TMR2 register is automatically cleared to 00h. The timer TMR2 Postscaler is incremented and its output is used to generate an interrupt if it is enabled.

The TMR2 and PR2 registers are both fully readable and writable. Counting may be stopped by clearing the TMR2ON bit, which contributes to power saving.

As a special option, the moment of TMR2 reset may be also used to determine synchronous serial communication baud rate.

The timer TMR2 is controlled by several bits of the T2CON register.

T2CON Register

		R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	Features
T2CON	-	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

-	Bit is unimplemented
R/W	Readable/Writable bit
(0)	After reset, bit is cleared

Fig. 4-14 T2CON Register

TOUTPS3 - TOUTPS0 - Timer2 Output Postcaler Select bits are used to determine the postscaler rate according to the following table:

TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	Postscaler Rate
0	0	0	0	1:1
0	0	0	1	1:2
0	0	1	0	1:3
0	0	1	1	1:4
0	1	0	0	1:5
0	1	0	1	1:6
0	1	1	0	1:7
0	1	1	1	1:8
1	0	0	0	1:9
1	0	0	1	1:10
1	0	1	0	1:11
1	0	1	1	1:12
1	1	0	0	1:13
1	1	0	1	1:14
1	1	1	0	1:15
1	1	1	1	1:16

Table 4-3 Postscaler Rate

TMR2ON - Timer2 On bit turns the timer TMR2 on.

- 1 - Timer T2 is on; and
- 0 - Timer T2 is off.

T2CKPS1, T2CKPS0 - Timer2 Clock Prescale bits determine prescaler rate:

T2CKPS1	T2CKPS0	Prescaler Rate
0	0	1:1
0	1	1:4
1	x	1:16

Table 4-4 Prescaler Rate

When using the TMR2 timer, one should know several specific details that have to do with its registers:

- Upon power-on, the PR2 register contains the value FFh;
- Both prescaler and postscaler are cleared by writing to the TMR2 register;
- Both prescaler and postscaler are cleared by writing to the T2CON register; and
- On any reset, both prescaler and postscaler are cleared.

Previous Chapter | Table of Contents | Next Chapter

Chapter 5: CCP Modules

The abbreviation CCP stands for *Capture/Compare/PWM*.

The CCP module is a peripheral which allows the user to time and control different events.

Capture Mode, allows timing for the duration of an event. This circuit gives insight into the current state of a register which constantly changes its value. In this case, it is the timer TMR1 register.

Compare Mode compares values contained in two registers at some point. One of them is the timer TMR1 register. This circuit also allows the user to trigger an external event when a predetermined amount of time has expired.

PWM - *Pulse Width Modulation* can generate signals of varying frequency and duty cycle.

The PIC16F887 microcontroller has two such modules - CCP1 and CCP2.

Both of them are identical in normal mode, with the exception of the Enhanced PWM features available on CCP1 only. This is why this chapter describes the CCP1 module in detail. Concerning CCP2, only the features distinguishing it from CCP1 will be covered.

Complicated? All this is only a simplified explanation on their operation. Everything is much more complicated in practice because these modules can operate in many different modes. Try to analyze their operation on the basis of the tables describing bit functions. If you use any CCP module, first select the mode you need, analyze the appropriate figure and then start changing bits of the registers or else...

CCP1 Module

A central part of this circuit is a 16-bit register CCPR1, which consists of the CCPR1L and CCPR1H registers. It is used for capturing or comparing with binary number stored in the timer register TMR1 (TMR1H and TMR1L).

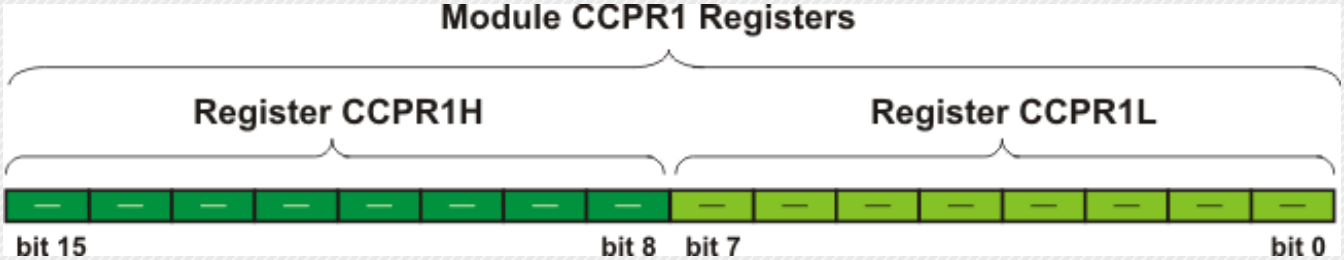


Fig. 5-1 CCP1 Module

In Compare mode, if enabled by software, the timer TMR1 reset may occur on match. Besides, the CCP1 module can generate PWM signals of varying frequency and duty cycle.

Bits of the CCP1CON register controls the CCP1 module.

CCP1 in Capture mode

In this mode, the timer register TMR1 (consisting of TMR1H and TMR1L) is copied to the CCP1 register (consisting of CCPR1H and CCPR1L) in the following situations:

- Every falling edge (1 » 0) on the RC2/CCP1 pin;
- Every rising edge (0 » 1) on the RC2/CCP1 pin;
- Every 4th rising edge (0 » 1) on the RC2/CCP1 pin; and
- Every 16th rising edge (0 » 1) on the RC2/CCP1 pin.

The combination of the four bits (CCP1M3 - CCP1M0) of the control register determines which of these events will trigger 16-bit data transfer. In addition, the following conditions must be met:

- RC2/CCP1 pin must be configured as input; and
- TMR1 module must operate as timer or synchronous counter.

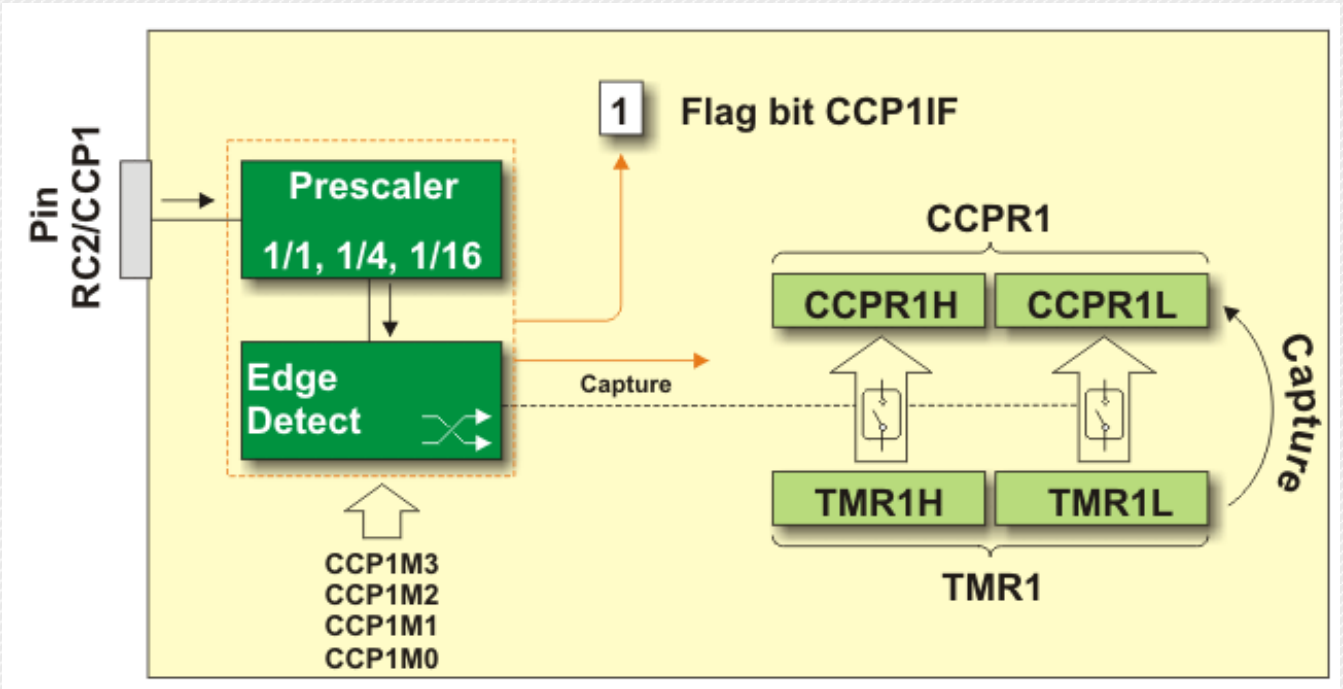


Fig. 5-2 CCP1 in Capture mode

The flag bit CCP1IF is set when a capture is made. If it happens and if the CCP1IE bit of the PIE register is set, then an interrupt occurs.

When the Capture mode is changed, an undesirable capture interrupts may be generated. In order to avoid that, both a

bit enabling CCP1IE interrupt and flag bit CCP1IF should be cleared prior to any change occuring in the control register.

Undesirable interrupt may be also generated by switching from one capture prescaler to another. To avoid this, the CCP1 module should be temporarily switched off before changing the prescaler.

The following program sequence is recommended:

```
BANKESEL CCP1CON
CLRF     CCP1CON     ;CONTROL REGISTER IS CLEARED
                     ;CCP1 MODULE IS OFF
MOVLW    XX          ;NEW PRESCALER MODE IS SELECTED
MOVWF    CCP1CON     ;NEW VALUE IS LOADED TO THE CONTROL REGISTER
                     ;CCP1 MODULE IS SIMULTANEOUSLY SWITCHED ON
```

CCP1 in Compare mode

In this mode, the value in the CCP1 register is constantly compared to the value in the timer register TMR1. When a match occurs, the output pin RC2/CCP1 logic state may be changed, which depends on the state of bits in the control register (CCP1M3 - CCP1M0). The flag-bit CCP1IF will be simultaneously set.

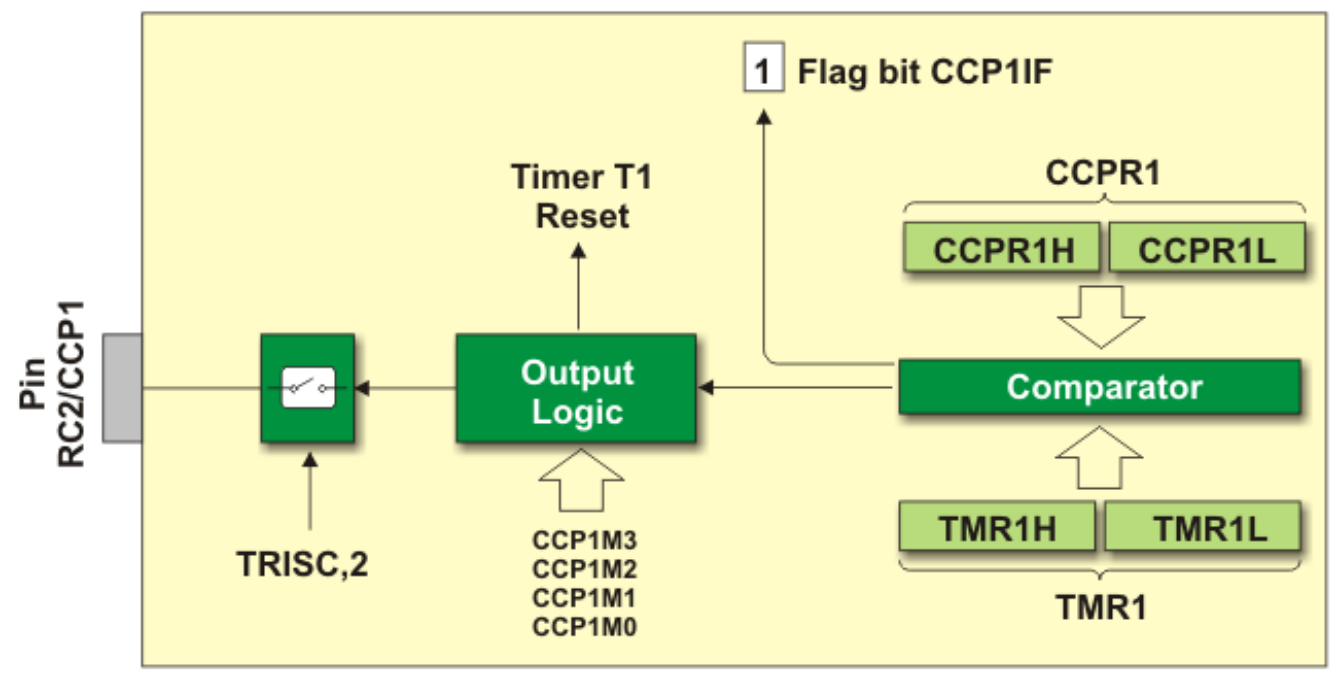


Fig. 5-3 CCP1 in Compare mode

To setup CCP1 module to operate in this mode, two conditions must be met:

- Pin RC2/CCP1 must be configured as output; and
- Timer TMR1 must be synchronized with internal clock.

CCP1 in PWM mode

Signals of varying frequency and duty cycle have a wide application in automation. A typical example is a power control circuit whose simple operation is shown in figure 5-4 below. If a logic zero (0) represents switch-off and logic one (1) represents switchon, the power that the load consumes will be directly proportional to the pulse duration. This ratio is often called *Duty Cycle*.

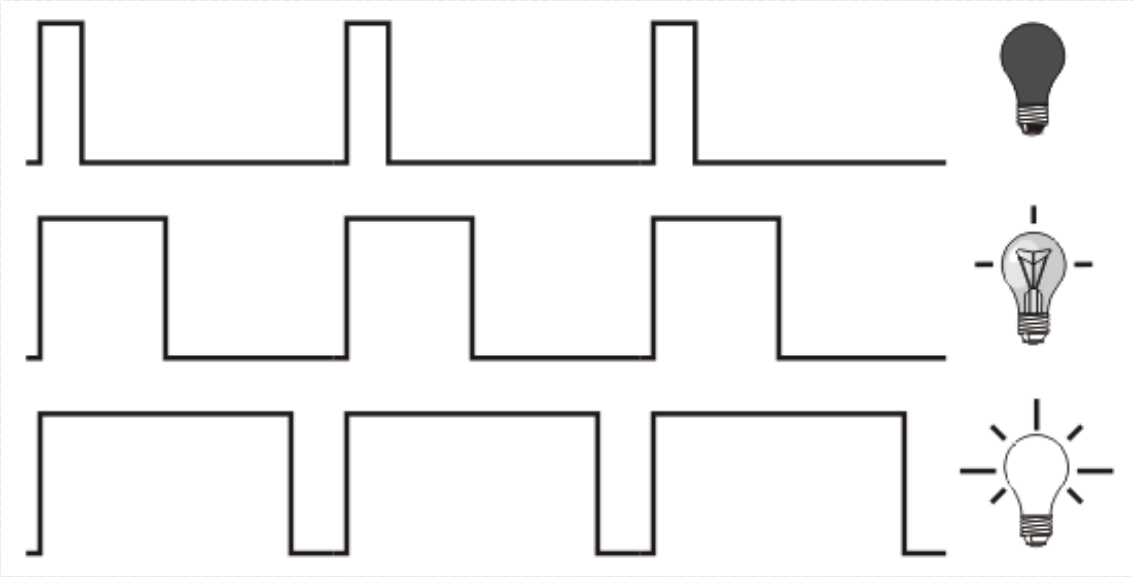


Fig. 5-4 CCP1 in PWM mode

Another example, common in practice, is the usage of PWM signals in the circuit for generating signals of arbitrary waveforms, for example, sinusoidal waveform. See figure 5-5 below:

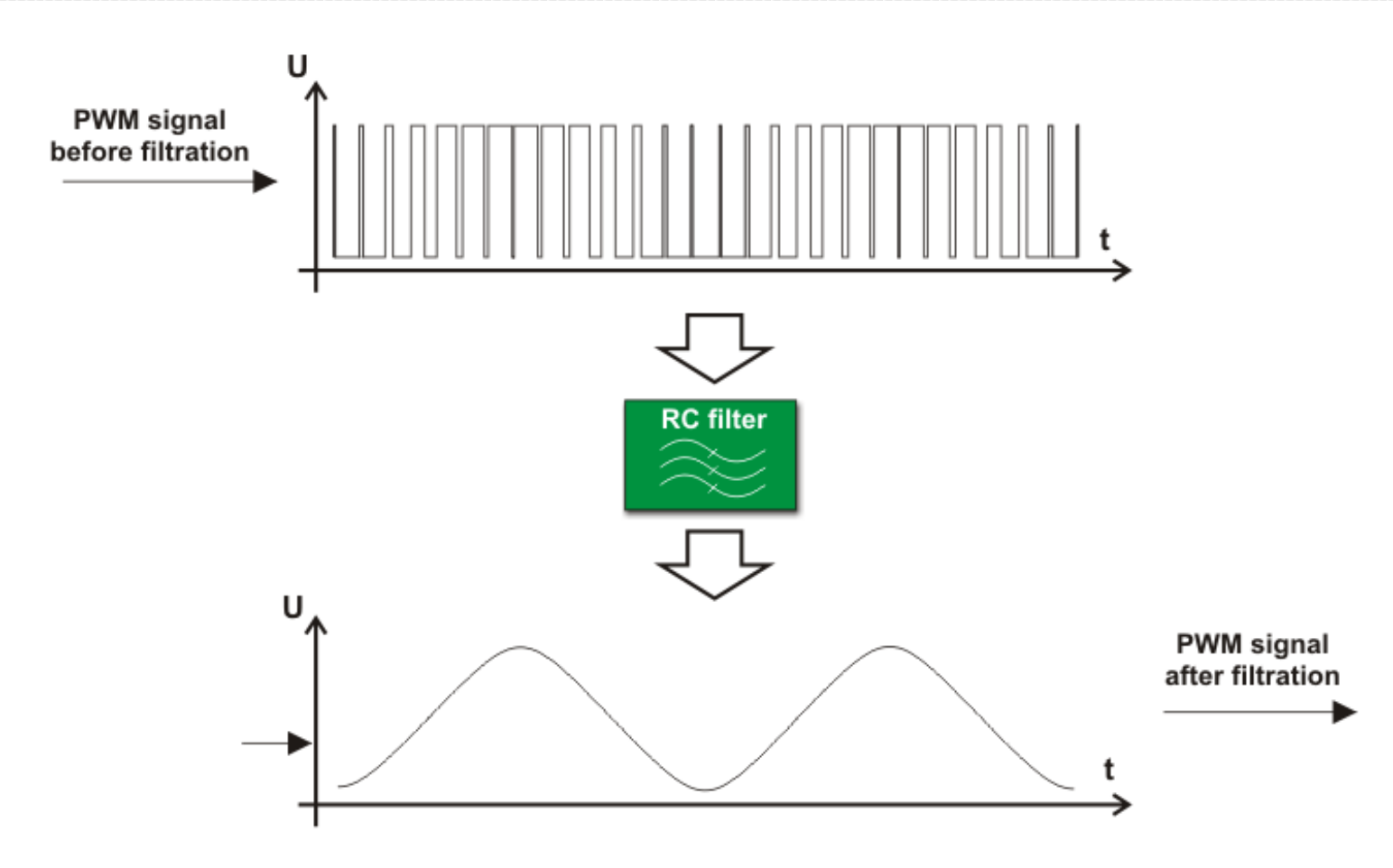


Fig. 5-5 CCP1 in PWM mode with filtration

Devices which operate in this way are often used in practice as switching regulators which control the operation of motors (speed, acceleration, deceleration etc.).

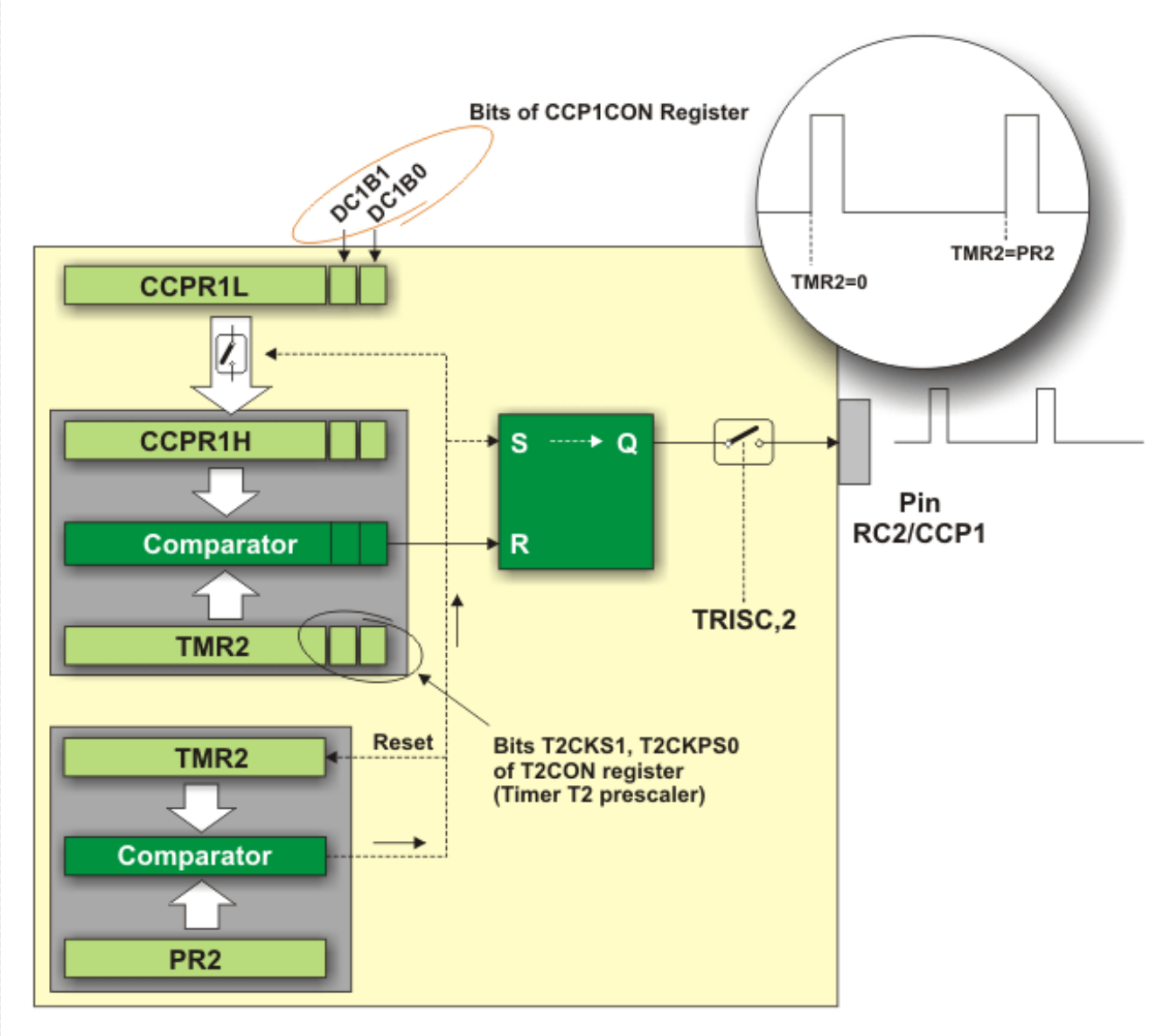


Fig. 5-6 PWM module

The figure above shows the block diagram of the CCP1 module setup in PWM mode. In order to generate a pulse of arbitrary form on its output pin, it is necessary to determine only two values- pulse frequency and duration.

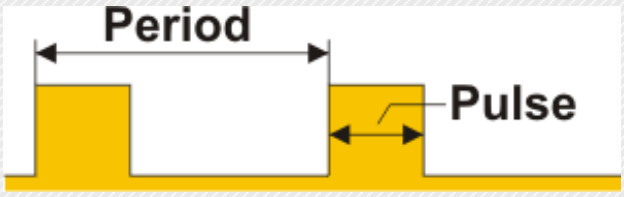


Fig.5-7 PWM Mode

PWM Period

The output pulse period (T) is specified by the PR2 register of the timer TMR2. The PWM period can be calculated using the following equation:

PWM Period(T) = (PR2 +1) * 4Tosc * TMR2 Prescale Value

If the PWM Period (T) is known then, it is easy to determine the signal frequency F because these two values are related

by equation F=1/T.

PWM Duty Cycle

The PWM duty cycle is specified by using in total of 10 bits: eight MSbs found in the CCPR1L register and two additional LSbs found in the CCP1CON register (DC1B1 and DC1B0). The result is 10-bit number contained in the formula:

Pulse Width = (CCPR1L,DC1B1,DC1B0) * Tosc * TMR2 Prescale Value

The following table shows how to generate PWM signals of varying frequency if the microcontroller uses 20 MHz quartz-crystal (Tosc=50nS).

Frequency [KHz]	1.22	4.88	19.53	78.12	156.3	208.3
TMR2 Prescaler	16	4	1	1	1	1
PR2 Register	FFh	FFh	FFh	3Fh	1Fh	17h

Table 5-1 PWM Duty Cycle

At last, two notes:

- Output pin will be constantly set in case the pulse width is by negligence determined to be larger than PWM period; and
- In this application, the timer TMR2 Postscaler cannot be used for generating longer PWM periods.

PWM Resolution

PWM signal is nothing more than the pulse sequence with varying duty cycle. For one specified frequency (number of pulses per second), there is a limited number of duty cycle combinations. This number is called resolution measured by bits. For example, a 10-bit resolution will result in 1024 discrete duty cycles, whereas an 8-bit resolution will result in 256 discrete duty cycles etc. In relation to this microcontroller, the resolution is specified by the PR2 register. The maximal value is obtained by writing number FFh.

PWM frequencies and resolutions (Fosc = 20MHz):

PWM Frequency	1.22kHz	4.88kHz	19.53kHz	78.12kHz	156.3kHz	208.3kHz
Timer Prescale	16	4	1	1	1	1
PR2 Value	FFh	FFh	FFh	3Fh	1Fh	17h
Maximum Resolution	10	10	10	8	7	6

Table 5-2 PWM Frequencies and Resolutions

PWM frequencies and resolutions (Fosc = 8MHz):

PWM Frequency	1.22kHz	4.90kHz	19.61kHz	76.92kHz	153.85kHz	200.0kHz
Timer Prescale	16	4	1	1	1	1
PR2 Value	65h	65h	65h	19h	0Ch	09h
Maximum Resolution	8	8	8	6	5	5

Table 5-3 PWM Frequencies and Resolutions

CCP1CON Register

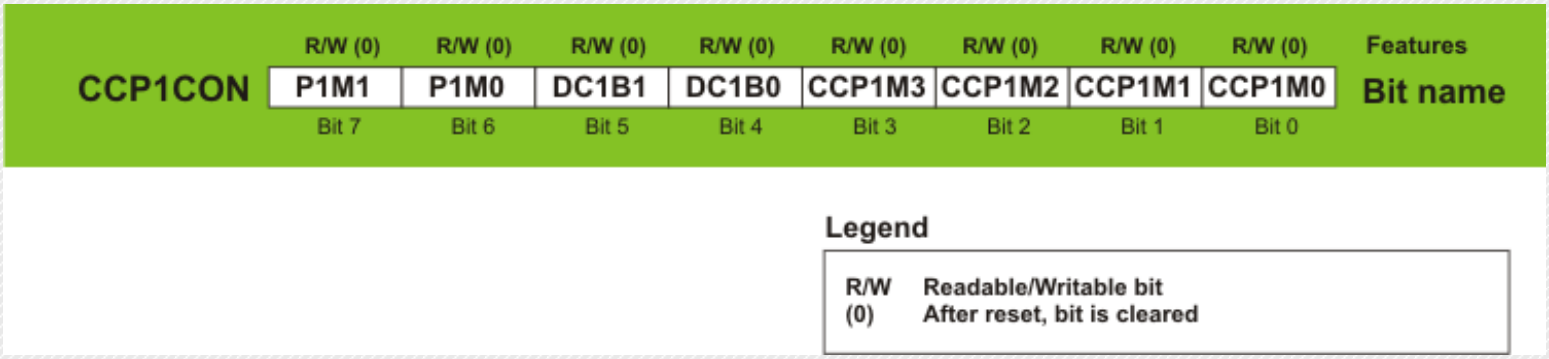


Fig. 5-8 CCP1CON Register

P1M1, P1M0 - PWM Output Configuration bits - In all modes, excepting PWM, the P1A pin is Capture/Compare module input. P1B, P1C and P1D pins act as input/output port D pins. In PWM mode, these bits affect the CCP1 module as shown in the table 5-4 below:

P1M1	P1M0	Mode
0	0	**PWM with single output**
		Pin P1A outputs modulated signal. Pins P1B, P1C and P1D are port D input/output
0	1	**Full Bridge - Forward configuration**
		Pin P1D outputs modulated signal Pin P1A is active Pins P1B and P1C are inactive
1	0	**Half Bridge configuration**
		Pins P1A and P1B output modulated signal PinsP1C and P1D are port D input/output
1	1	**Full Bridge - Reverse configuration**
		Pin P1B outputs modulated signal Pin P1C is active Pins P1A and P1D are inactive

Table 5-4 CCP1CON Register

DC1B1, DC1B0 - PWM Duty Cycle Least Significant bits - are only used in PWM mode in which they represent two least significant bits of a 10-bit number. This number determines PWM signal's duty cycle. The rest of bits (8 in total) are stored in the CCPR1L register.

CCP1M3 - CCP1M0 - CCP1 Mode Select bits determine the mode of the CCP1 module.

CCP1M3	CCP1M2	CCP1M1	CCP1M0	Mode
0	0	0	0	Module is disabled (reset)
0	0	0	1	Unused
0	0	1	0	**Compare mode**
				CCP1IF bit is set on match
0	0	1	1	Unused
0	1	0	0	**Capture mode**
				Every falling edge on the CCP1 pin
0	1	0	1	**Capture mode**
				Every rising edge on the CCP1 pin
0	1	1	0	**Capture mode**
				Every 4th rising edge on the CCP1 pin
0	1	1	1	**Capture mode**

				Every 16th rising edge on the CCP1 pin
1	0	0	0	**Compare mode** Output and CCP1IF bit are set on match
1	0	0	1	**Compare mode** Output is cleared and CCP1IF bit is set on match
1	0	1	0	**Compare mode** Interrupt request arrives and bit CCP1IF is set on match
1	0	1	1	**Compare mode** Bit CCP1IF is set and timers 1 or 2 registers are cleared
1	1	0	0	**PWM mode** Pins P1A and P1C are active-high Pins P1B and P1D are active-high
1	1	0	1	**PWM mode** Pins P1A and P1C are active-high Pins P1B and P1D are active-low
1	1	1	0	**PWM mode** Pins P1A and P1C are active-low Pins P1B and P1D are active-high
1	1	1	1	**PWM mode** Pins P1A and P1C are active-low Pins P1B and P1D are active-low

Table 5-5 Modes of Operations

CCP2 Module

Excluding the different names of registers and bits, this module is a very good copy of the CCP1 module setup in normal mode (previously discussed). There is only one true difference between their modes when CCP2 operates in Compare mode.

That difference refers to the timer T1 reset signal. Namely, if A/D converter is enabled at the moment the values of the TMR1 and CCPR2 registers match, the timer T1 reset signal will automatically start A/D conversion.

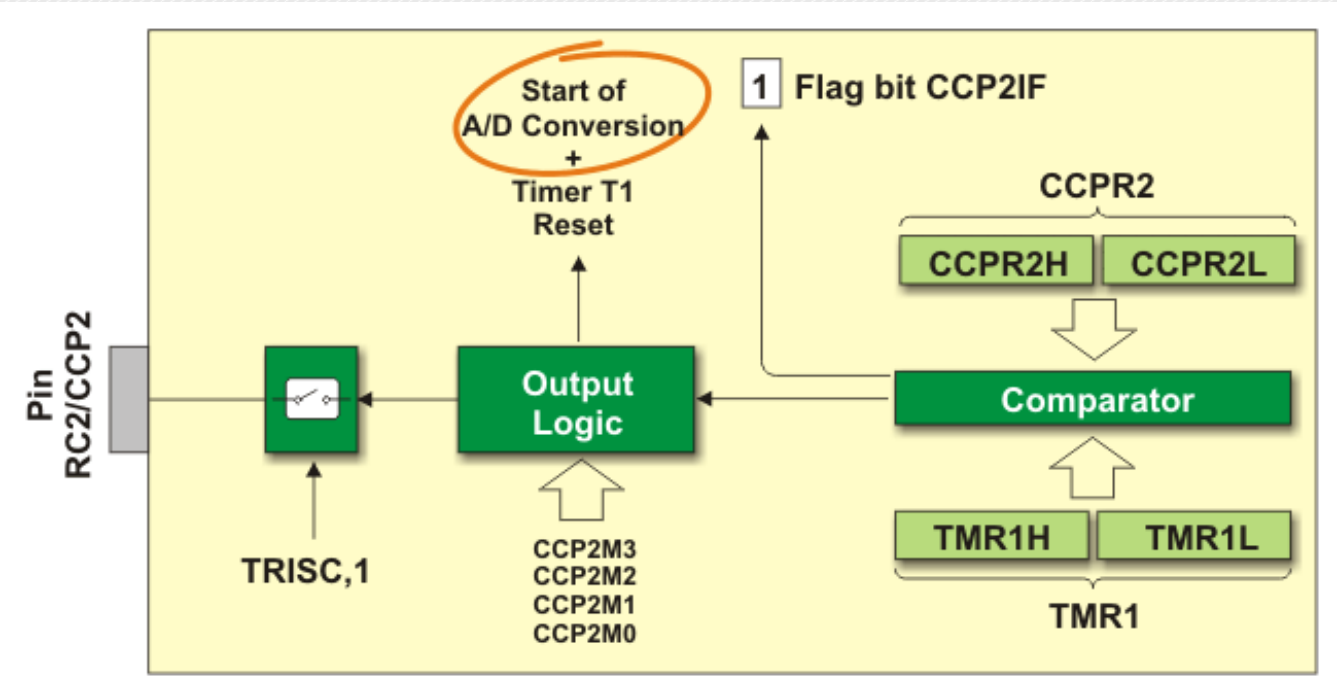


Fig. 5-9 CCP2 Module

Similar to the pervious module, this circuit is under control of the bits of the control register. This time, it is the CCP2CON register.

CCP2CON Register

			R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	Features
CCP2CON	-	-	DC2B1	DC2B0	CCP2M3	CCP2M2	CCP2M1	CCP2M0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

-	Bit is unimplemented
R/W	Readable/Writable bit
(0)	After reset, bit is cleared

Fig. 5-10 CCP2CON Register

DC2B1, DC2B0 - PWM Duty Cycle Least Significant bits - are only used in PWM mode representing two least significant bits of a 10-bit number. This number determines PWM signal's duty cycle. The rest of bits (8 in total) are stored in the CCPR2L register.

CCP2M3 - CCP2M0 - CCP2 Mode Select bits select CCP2 mode.

CCP2M3	CCP2M2	CCP2M1	CCP2M0	Mode
0	0	0	0	Module is disabled (reset)
0	0	0	1	Unused
0	0	1	0	Unused
0	0	1	1	Unused
0	1	0	0	**Capture mode**
				Every falling edge on the CCP2 pin
0	1	0	1	**Capture mode**
				Every raising edge on the CCP2 pin
0	1	1	0	**Capture mode**
				Every 4th rising edge on the CCP2 pin
0	1	1	1	**Capture mode**
				Every 16th rising edge on the CCP2 pin
1	0	0	0	**Compare mode**
				Output and CCP2IF bit are set on match
1	0	0	1	**Compare mode**
				Output is cleared and CCP2IF bit is set on match
1	0	1	0	**Compare mode**
				Interrupt is generated, CCP2IF bit is set and CCP2 pin is unaffected on match
1	0	1	1	**Compare mode**
				CCP2IF bit is set, Timer 1 registers are cleared, A/D conversion is started if the A/D converter is on on match
1	1	x	x	**PWM mode**

Table 5-6 CCP2CON Register

In short: Setup CCP1 module for PWM operation

In order to setup the CCP module for PWM operation, the following steps should be taken:

- Disable the CCP1 output pin. It should be configured as input;
- Set the PWM period by loading the PR2 register;
- Configure the CCP module for the PWM mode by combining bits of the CCP1CON register;
- Set the PWM signal's duty cycle by loading the CCPR1L register and using bits DC1B1 and DC1B0 of the CCP1CON register;
- Configure and start timer TMR2:
  - Clear the TMR2IF interrupt flag bit of the PIR1 register;
  - Set the timer TMR2 prescale value by loading bits T2CKPS1 and T2CKPS0 of the T2CON register;
  - Start the timer TMR2 by setting the TMR2ON bit of the T2CON register;

- Enable PWM output pins after one PWM cycle has been finished:
  - Wait for the timer TMR2 overflow (TMR2IF bit of the PIR1register is set); and
  - Configure the appropriate pin as output by clearing bit of the TRIS register.

CCP1 in Enhanced Mode

The enhanced mode is available on CCP1 only. Basically, this module does not differ from the one previously described and enhancement refers to transmission of PWM signal to the output pins. Why is it so important? Because the microcontrollers are more frequently used in control systems for electric motors. These devices are not described here, but if you ever have had a chance to work on development of similar devices, you will recognize elements which, until quite recently, have been used as external ones. Normally, all these elements are now integrated into the microcontroller and can operate in several different modes.

Single Output PWM Mode

This mode is enabled only in the event that the P1M1 and P1M0 bits of the CCP1CON register are cleared. In this case, there is only one PWM signal which can be simultaneously available on a maximum of four different output pins. Besides, the PWM signal may appear in basic or inverted waveform. Signal distribution is determined by the bits of the PSTRCON register, while it's polarity is determined by the CCP1M1 and CCP1M0 bits of the CCP1CON register.

When an inverted output is in use, the pins are low-active and pulses having the same waveform are always generated in pairs: on the P1A and P1C pins and P1B and P1D pins, respectively.

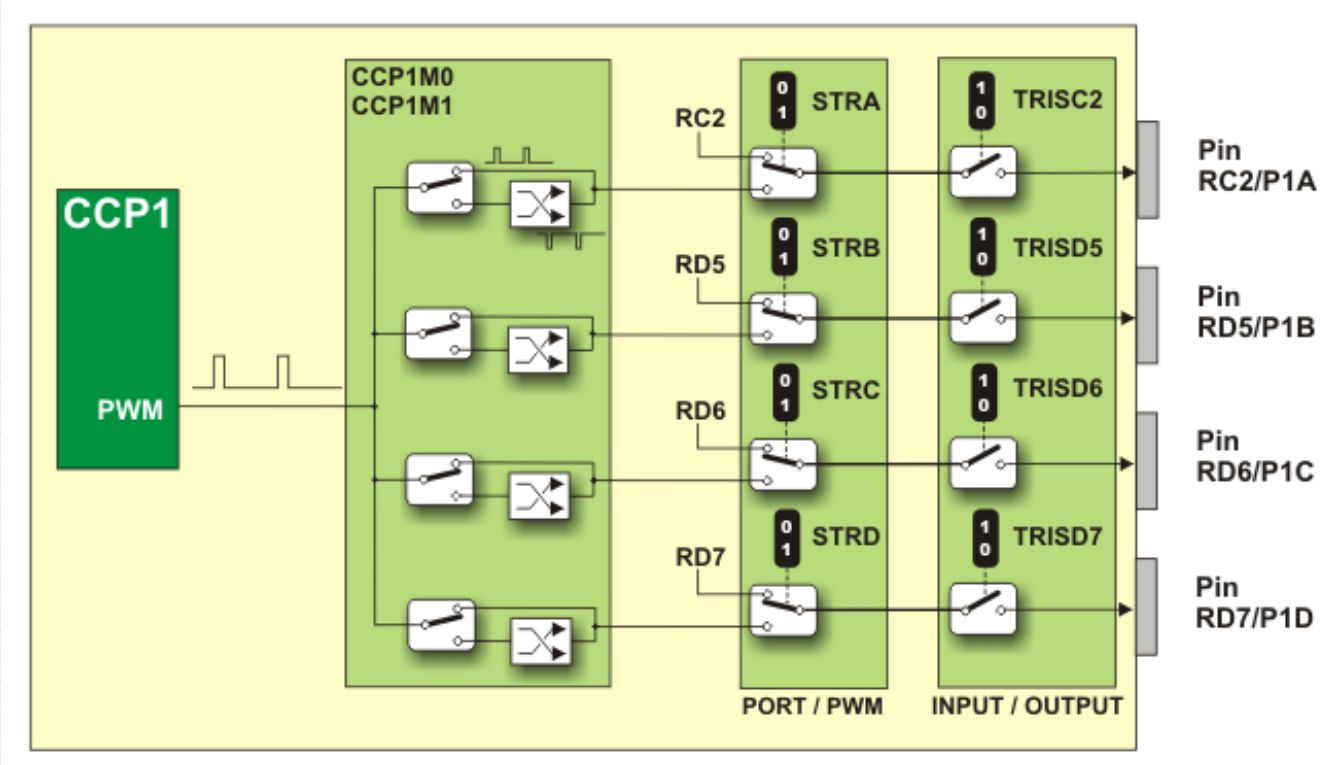


Fig. 5-11 Single Output PWM Mode

Half-Bridge Mode

In this mode, the PWM signal is output on the P1A pin, while at the same time the complementary PWM signal is output on the P1B pin. Such pulses activate MOSFET drivers in Half-Bridge mode which enable/disable current flow through device.

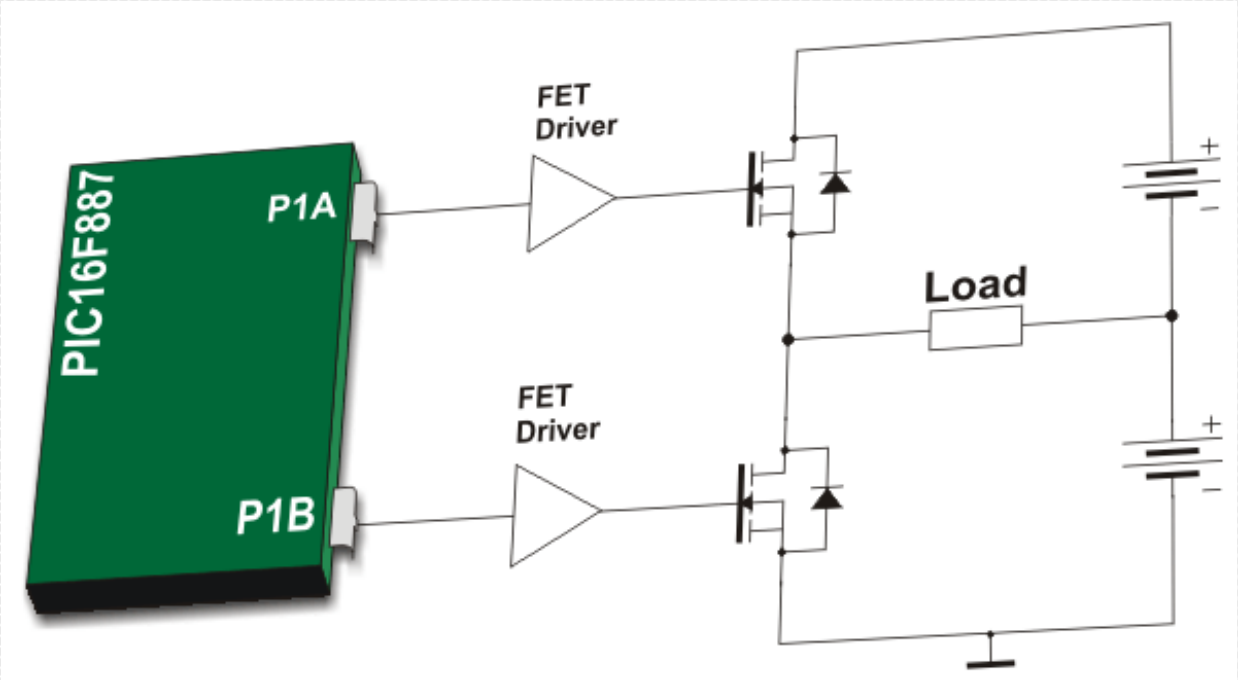


Fig. 5-12 Half-Bridge Mode

In relation to this circuit, it is very dangerous to switch on both MOSFET drivers simultaneously. The short circuit caused in that moment will be fatal. In order to avoid that, it is necessary to provide a short delay between switching drivers on and off. This delay is marked as "td" in figure 5-13 below. The problem is solved by using the PDC0-PDC6 bits of the PWM1CON register.

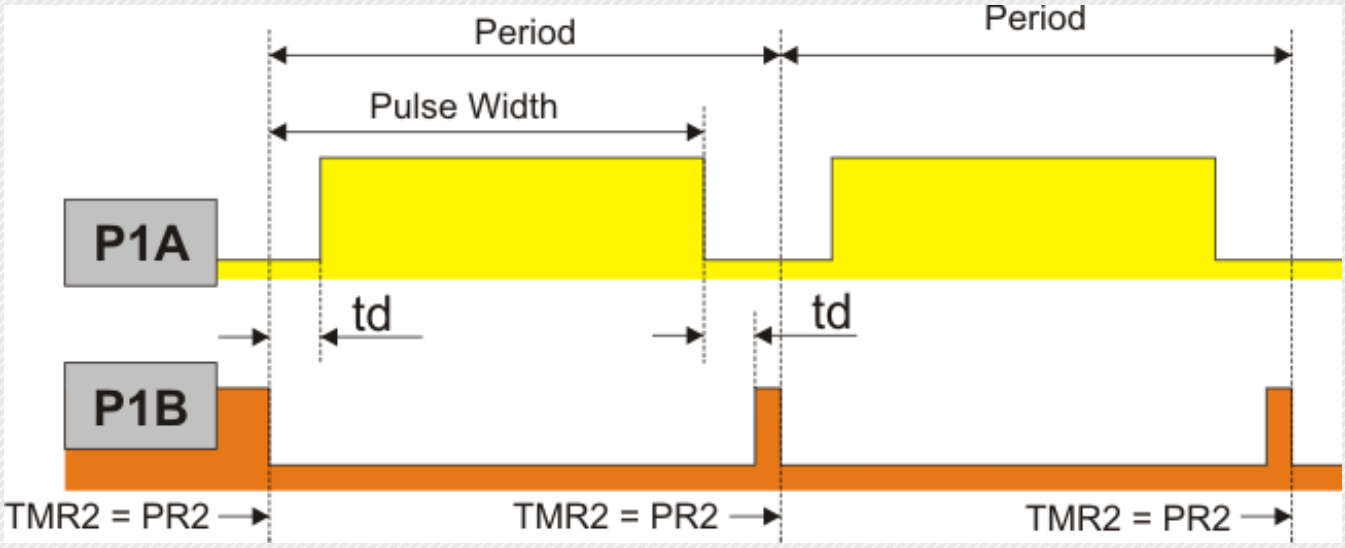


Fig. 5-13 Half Bridge Mode

As shown in figure 5-14, the same mode can be used to activate MOSFET drivers in Full Bridge:

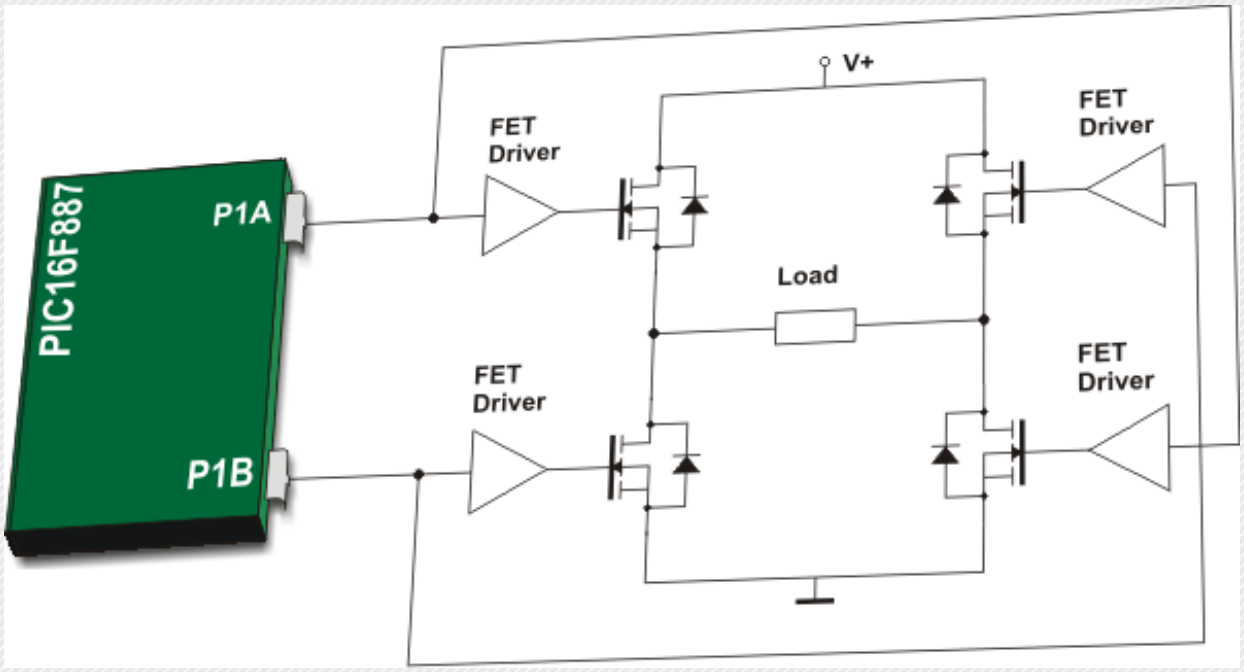


Fig. 5-14 Activate MOSFET drivers

Full-Bridge Mode

In Full-Bridge mode, all four pins are used as outputs. In practice, this mode is commonly used to run motors, which provides simple and complete control of speed and rotation direction. There are two such configurations: *Full Bridge-Forward* and *Full Bridge-Reverse*.

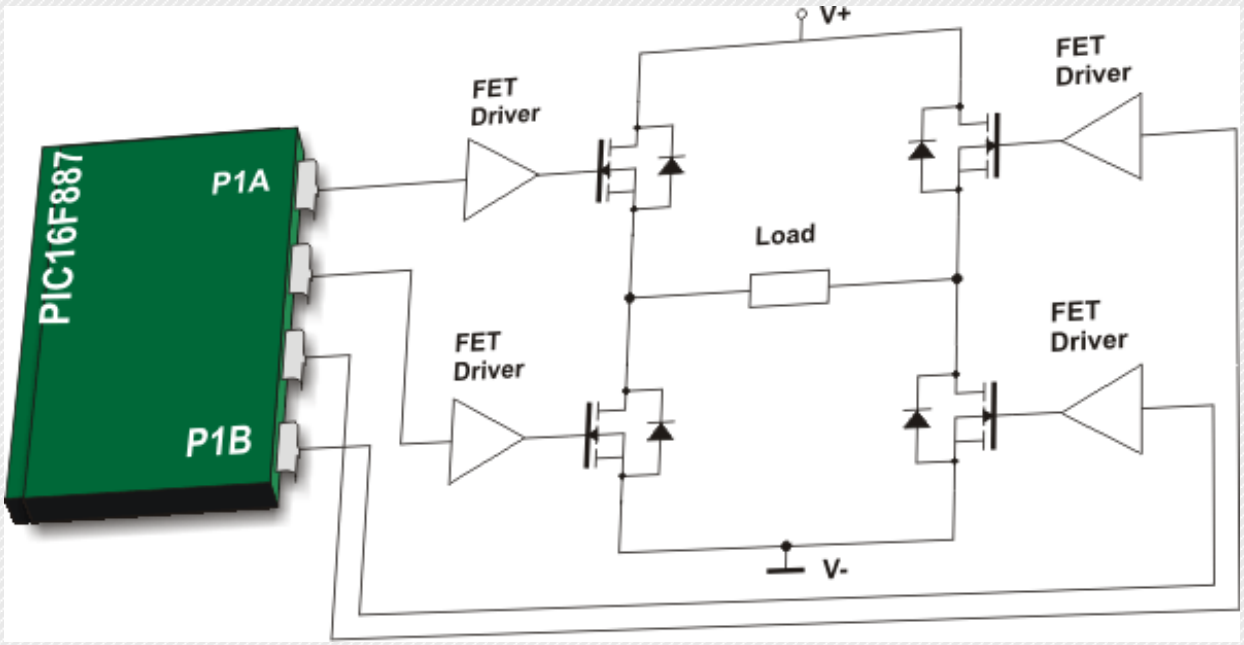


Fig. 5-15 Full-Bridge Mode

Full Bridge - Forward Configuration

In *Forward* mode the following occurs:

- Logic one (1) appears on the P1A pin (pin is high-active);
- Pulse sequence appears on the P1D pin; and
- Logic zero (0) appears on the P1B and P1C pins (pins are low-active).

Figure below shows the state of the P1A-P1D pins during one full PWM cycle.

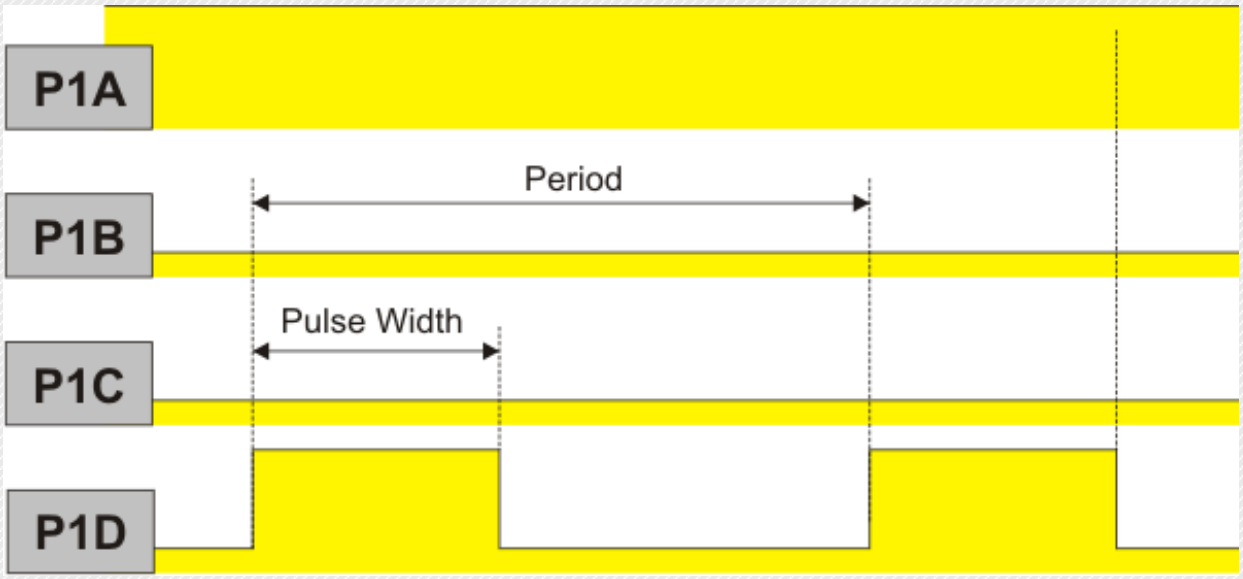


Fig. 5-16 Forward Mode

Full Bridge - Reverse Configuration

The same occurs in *Reverse* mode, except of the pins functions:

- Logic one (1) appears on the P1C pin (pin is active-high);
- Pulse sequence appears on the P1B pin; and
- Logic zero (0) appears on the P1A and P1D pins (pins are active-low).

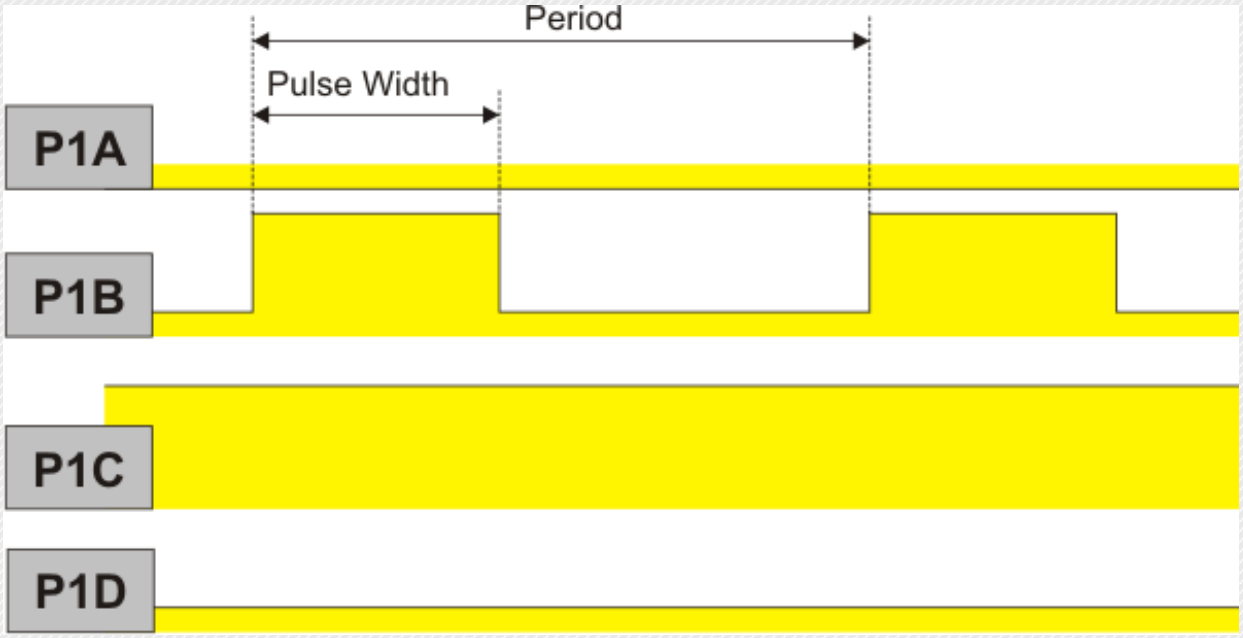


Fig. 5-17 Reverse Mode

PWM1CON Register STRC PWM Restart Enable bit

- 1 - Upon auto-shutdown, the PWM module is automatically reset, while the ECCPASE bit of the ECCPAS register is cleared.
- 0 - In order to restart PWM module upon auto-shutdown, the ECCPASE bit must be cleared in software.

PDC6 - PDC0 - PWM Delay Count bits. 7-digit binary number determines the number of instruction cycles (4*Tosc) added as time delay during the activation of PWM output pins.

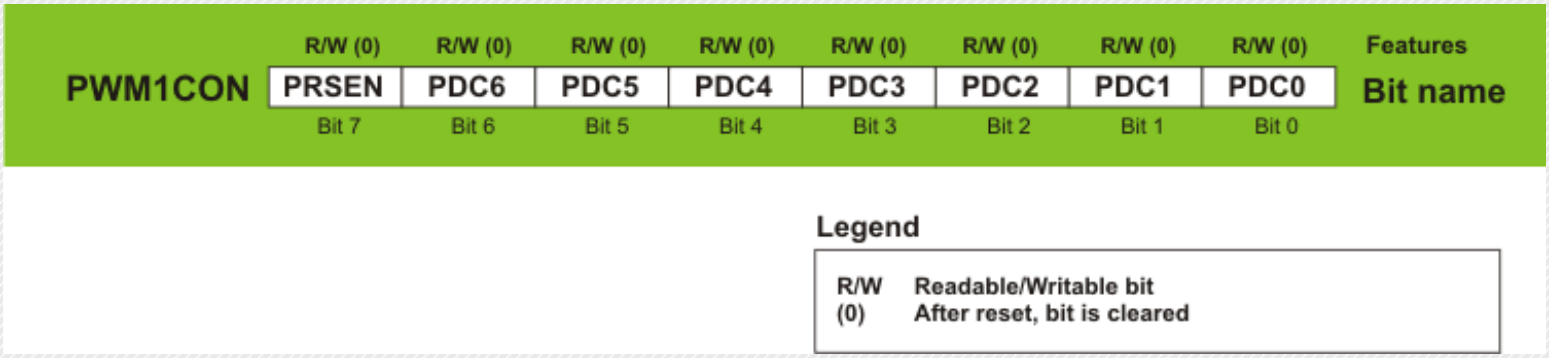


Fig. 5-18 PWM1CON Register

PSTRCON Register

STRSYNC - Steering Sync bit determines the moment of PWM pulse steering:

- 1 - Steering occurs upon the PSTRCON has been changed, but only if a PWM waveform is completed; and
- 0 - Steering occurs upon the PSTRCON register has been changed. The PWM signal on output pin is immediately changed with no regard to whether the previous cycle is completed or not. This operation is useful when it is needed to immediately remove a PWM signal from the pin.

STRD - Steering Enable bit D determines the P1D pin function.

- 1 - P1D pin has the PWM waveform with polarity controlled by the CCP1M0 and CCP1M1 bits; and
- 0 - Pin is configured as general Port D input/output.

STRC Steering Enable bit C determines the P1C pin function.

- 1 - P1C pin has the PWM waveform with polarity controlled by the CCP1M0 and CCP1M1 bits; and
- 0 - Pin is configured as general port D input/output.

STRB - Steering Enable bit B determines the P1B pin function.

- 1 - P1B pin has the PWM waveform with polarity controlled by the CCP1M0 and CCP1M1 bits; and
- 0 - Pin is configured as general port D input/output.

STRA - Steering Enable bit A determines the P1A pin function.

- 1 - P1D pin has the PWM waveform with polarity controlled by the CCP1M0 and CCP1M1 bits; and
- 0 - Pin is configured as general port Ainput/output.

ECCPAS Register

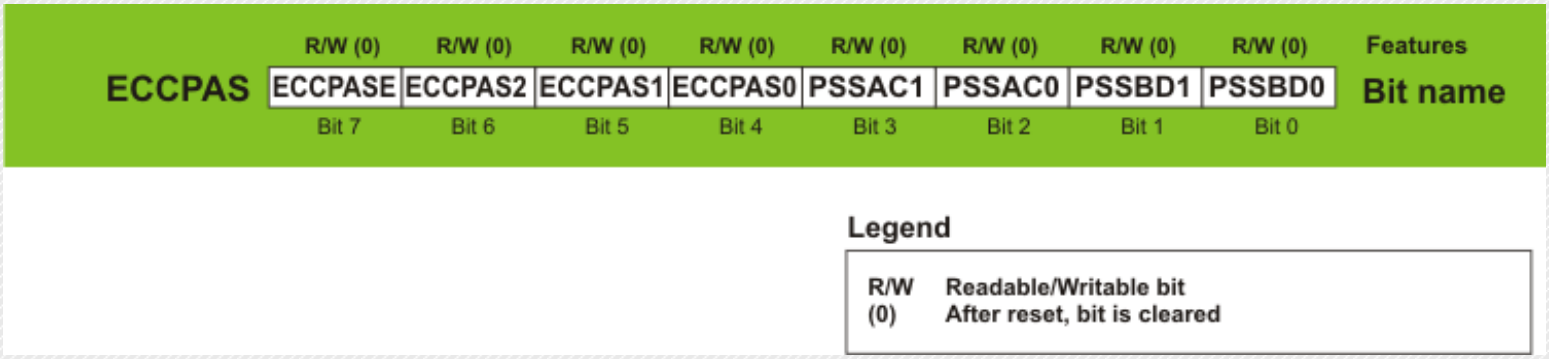


Fig. 5-19 ECCPAS Register

ECCPASE - ECCP Auto-Shutdown Event Status bit indicates whether shut-down of CCP module has occurred (Shutdown state):

- 1 - CCP module is in Shutdown state; and
- 0 - CCP module operates normally.

ECCPAS2 - ECCPAS0 - ECCP Auto-Shutdown Source Select bits select auto shutdown source:

ECCPAS2	ECCPAS1	ECCPAS0	Shutdown state source
0	0	0	Shutdown state disabled
0	0	1	Comparator C1 output change
0	1	0	Comparator C2 output change
0	1	1	Comparator C1 or C2 output change
1	0	0	Logic zero (0) on INT pin
1	0	1	Logic zero (0) on INT pin or comparator C1 output change
1	1	0	Logic zero (0) on INT pin or comparator C2 output change
1	1	1	Logic zero (0) on INT pin or comparator C1 or C2 output change

Table 5-7 ECCPAS Register

PSSAC1, PSSAC0 - Pins P1A, P1C Shutdown State Control bits define logic state on output pins P1A and P1C when CCP module is in shutdown state.

PSSAC1	PSSAC0	Pins logic state
0	0	0
0	1	1
1	X	High impedance (Tri-state)

Table 5-8 A&C Logic States

PSSBD1, PSSBD0 - Pins P1B, P1D Shutdown State Control bits define logic state on output pins P1B and P1D when CCP module is in shutdown state.

PSSBD1	PSSBD0	Pins logic state
0	0	0
0	1	1
1	X	High impedance (Tri-state)

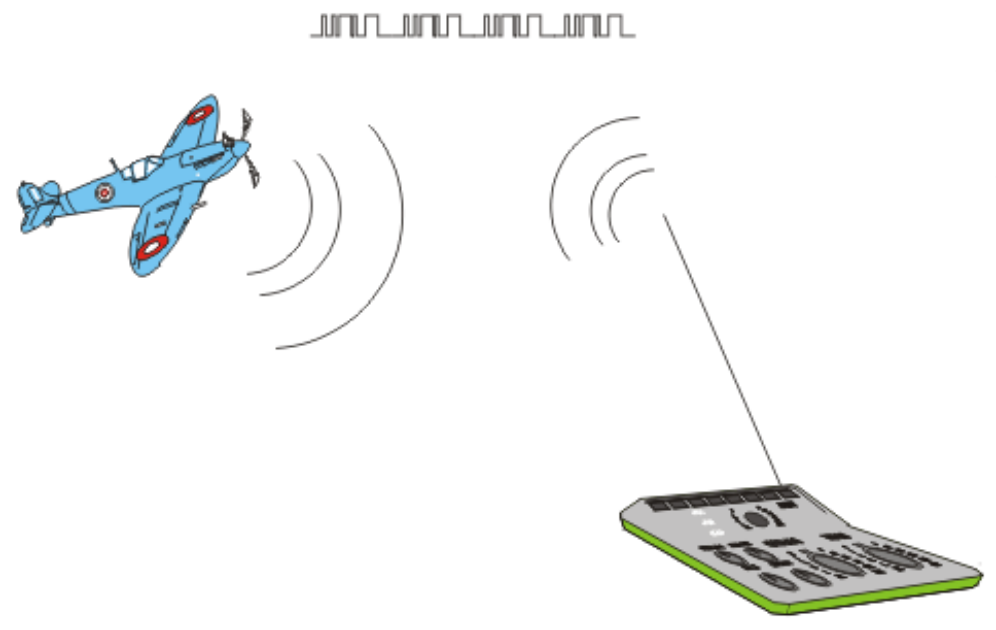
Table 5-9 B&D Logic States

Previous Chapter | Table of Contents | Next Chapter

Chapter 6: Serial Communication Modules

EUSART

The Enhanced Universal Synchronous Asynchronous Receiver Transmitter (EUSART) module is a serial I/O communication peripheral. It is also known as Serial Communications Interface (SCI). It contains all clock generators, shift registers and data buffers necessary to perform an input or output serial data transfer independently of the device program execution. As its name states, apart from the usage of clock for synchronization, this module can also establish asynchronous connection, which makes it irreplaceable in some applications.



For example, in the event that it is difficult or impossible to provide special channels for clock and data transfer (for example, radio remote control or infrared), the EUSART module presents itself as a convenient solution.

Fig. 6-1 Remote Control and Plane

The EUSART system integrated into the PIC16F887 microcontroller has the following features:

- *Full-duplex* asynchronous transmit and receive;
- Programmable 8- or 9-bit character length;
- Address detection in 9-bit mode;
- Input buffer overrun error detection; and
- Half-duplex communication in synchronous mode (master or slave).

EUSART Asynchronous Mode

The EUSART transmits and receives data using standard non-return-to-zero (NRZ) format. As seen in figure 6-2 below, this mode does not use clock signal, while the data format being transferred is very simple:

Fig. 6-2 EUSART Asynchronous Mode

Briefly, each data is transferred in the following way:

- In idle state, data line has high logic level (1);
- Each data transmission starts with START bit which is always a zero (0);
- Each data is 8- or 9-bit wide (LSB bit is first transferred); and
- Each data transmission ends with STOP bit which always has logic level which is always a one (1).

EUSART Asynchronous Transmitter

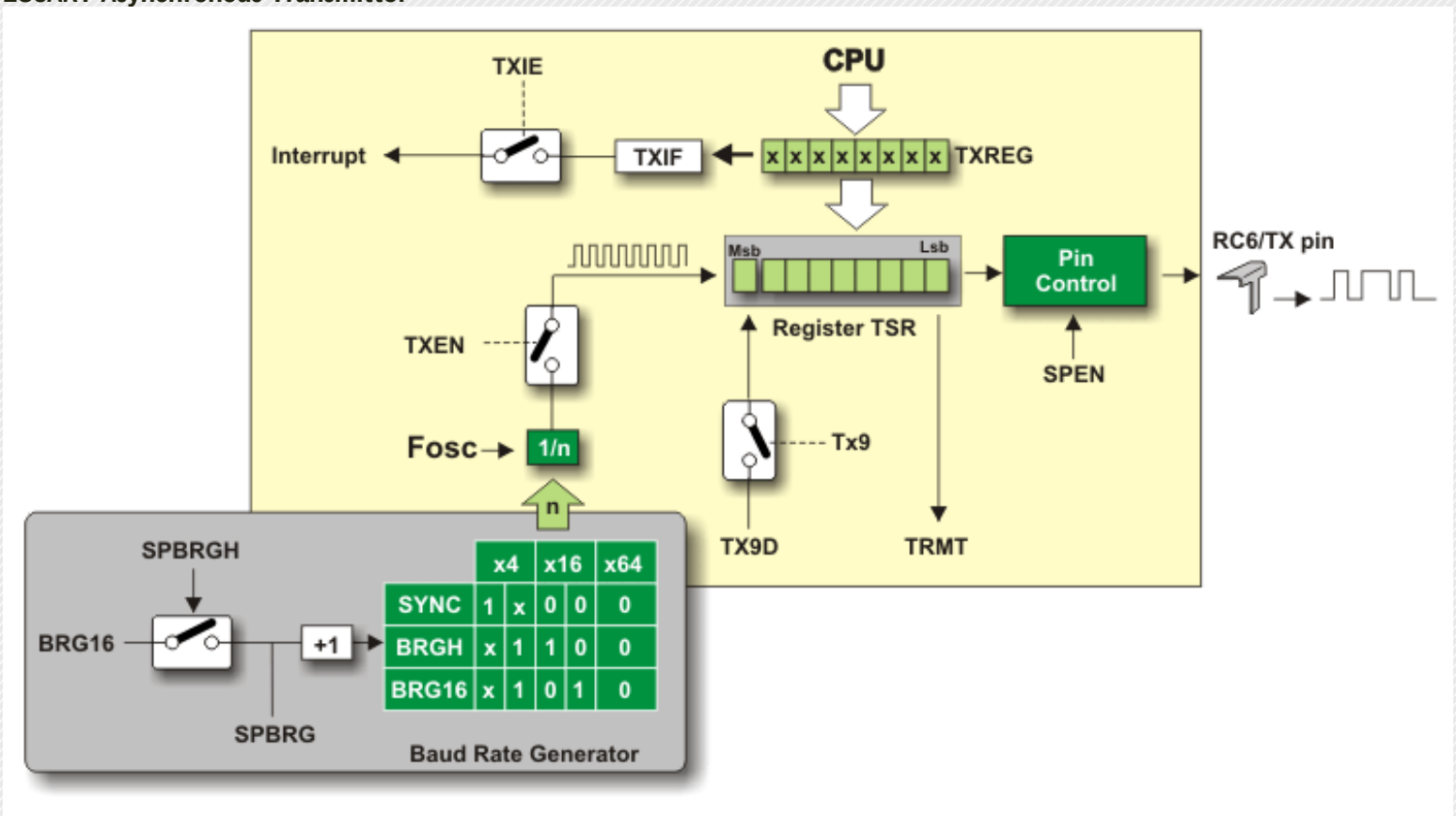


Fig. 6-3 EUSART Asynchronous Transmitter

In order to enable data transmission via EUSART module, it is necessary to configure it to operate as a transmitter. In other words, it is necessary to define the state of the following bits:

TXEN = 1 - EUSART transmitter is enabled by setting this bit of the TXSTA register;

SYNC = 0 - EUSART is configured to operate in asynchronous mode by clearing this bit of the TXSTA register; and

SPEN = 1 - By setting this bit of the RCSTA register, EUSART is enabled and the TX/CK pin is automatically configured as output. If this bit is simultaneously used for some analog function, it must be disabled by clearing the corresponding bit of the ANSEL register.

The central part of the EUSART transmitter is the shift register TSR which is not directly accessible by the user. In order to start transmission, the module must be enabled by setting the TXEN bit of the TXSTA register. Data to be sent should be written to the TXREG register, which will cause the following sequence of events:

- Byte will be immediately transferred to the shift register TSR;
- TXREG register remains empty, which is indicated by setting flag bit TXIF of the PIR1 register. If the TXIE bit of the PIE1 register is set, an interrupt will be generated. Besides, the flag is set regardless of whether an interrupt is enabled or not. Also, it cannot be cleared by software, but by writing new data to the TXREG register;
- Control electronics "pushes" data toward the TX pin in rhythm with internal clock: START bit (0) ... data ... STOP bit (1);
- When the last bit leaves the TSR register, the TRMT bit of the TXSTA regis ter is automatically set; and
- If the TXREG register has received a new character data in the meantime, the whole procedure is repeated immediately after the STOP bit of the previous character has been transmitted.

Sending 9-bit data is enabled by setting the TX9 bit of the TXSTA register. The TX9D bit of the TXSTA register is the ninth and Most Significant data bit. When transferring 9-bit data, the TX9D data bit must be written before writing the 8 least significant bits into the TXREG register. All nine bits of data will be transferred to the TSR shift register immediately after the TXREG write is complete.

EUSART Asynchronous Receiver

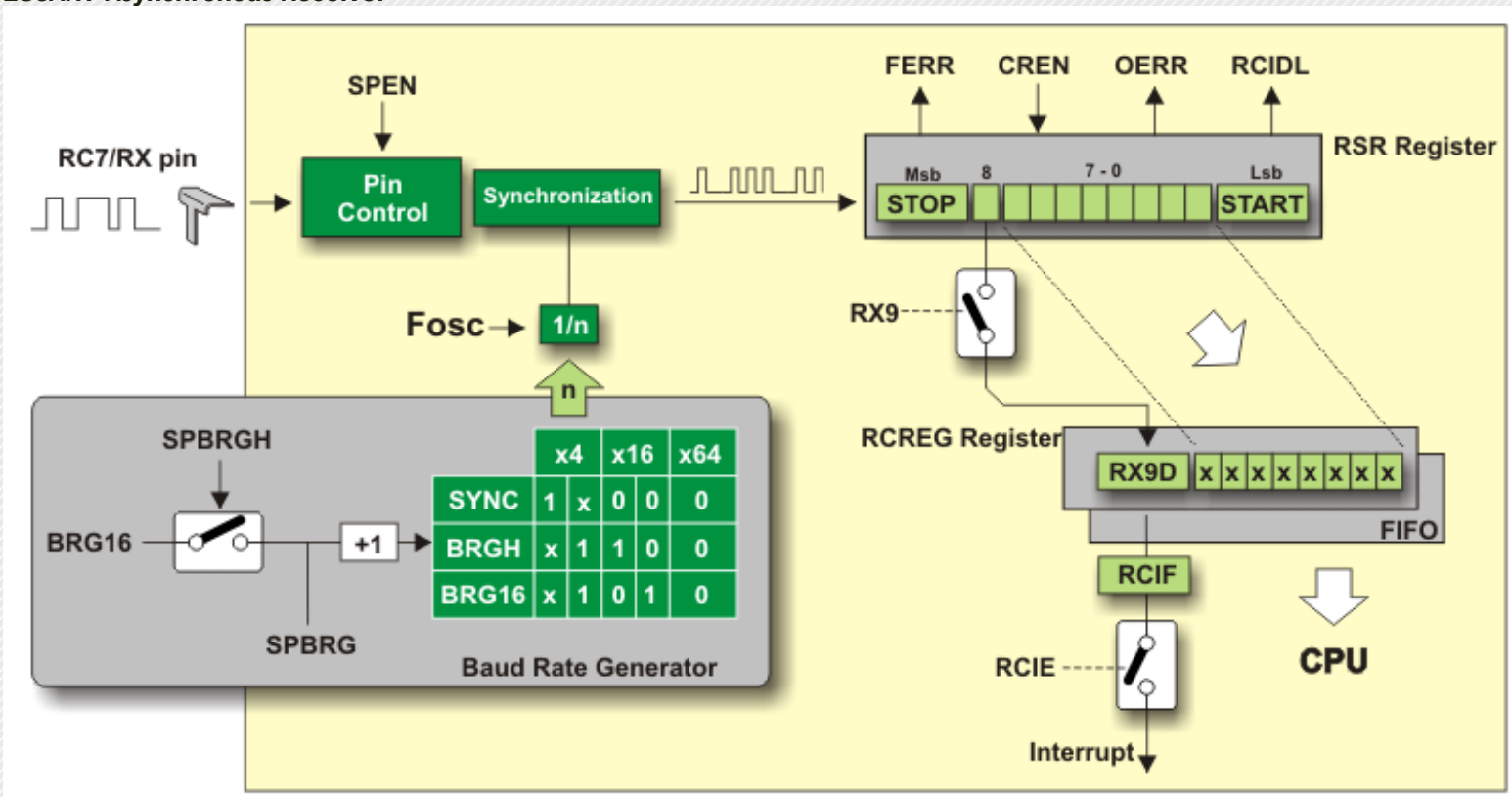


Fig. 6-4 EUSART Asynchronous Receiver

Similar to the activation of EUSART transmitter, in order to enable the receiver it is necessary to define the following bits:

CREN = 1 - EUSART receiver is enabled by setting this bit of the RCSTA register;

SYNC = 0 - EUSART is configured to operate in asynchronous mode by clearing this bit stored in the TXSTA register; and

SPEN = 1 - By setting this bit of the RCSTA register, EUSART is enabled and the RX/DT pin is automatically configured as input. If this bit is simultaneously used for some analog function, it must be disabled by clearing the corresponding bit of the ANSEL register.

When this first and necessary step is accomplished and START bit is detected, data is transferred to the shift register RSR through the RX pin. When the STOP bit has been received, the following occurs:

- Data is automatically transferred to the RCREG register (if empty);
- The flag bit RCIF is set and an interrupt, if enabled by the RCIE bit of the PIE1 register, occurs. Similar to transmitter, the flag bit is cleared by software only, i.e. by reading the RCREG register. Bear in mind that this is a two character FIFO memory (*first-in, first-out*) which allows reception of two characters simultaneously;
- If the RCREG register is occupied (contains two bytes) and the shift register detects new STOP bit, the overflow bit OERR will be set. In this case, a new coming data is lost, and the OEER bit must be cleared by software. It is done by clearing and resetting the CREN bit.
  Note: it is not possible to receive new data as far as the OERR bit is set;
- If the STOP bit is zero (0), the FERR bit of the RCSTAregister detecting receive error will be set; and
- To receive 9-bit data it is necessary to set the RX9 bit of the RCSTA register.

Receive Error Detection

There are two types of errors which the microcontroller can automatically detect. The first one is called *Framing error* and occurs when the receiver does not detect the STOP bit at the expected time. Such error is indicated via the FERR bit of the RCSTA register. If this bit is set, it means that the last received data may be incorrect. It is important to know several things:

- A *Framing error* does not generate an interrupt by itself;
- If this bit is set, the last received data has an error;
- A framing error (bit set) does not prevent reception of new data;
- The FERR bit is cleared by reading received data, which means that check must be done before data reading; and
- The FERR bit cannot be cleared by software. If needed, it can be cleared by clearing the SPEN bit of the RCSTA register. It will simultaneously cause reset of the whole EUSART system.

Another type of error is called *Overrun Error*. The receive FIFO can hold two characters. An overrun error will be generated if the third character is received. Simply, there is no space for another one byte and an error is unavoidable! When this happens the OERR bit of the RCSTA register is set. The consequences are the following:

- Data already stored in the FIFO registers (two bytes) can be normally read;
- No additional data will be received until the OERR bit is cleared; and
- This bit is not directly accessed. To clear it, it is necessary to clear the CREN bit of the RCSTA register or to reset the whole EUSART system by clearing the SPEN bit of the RCSTA register.

Receiving 9-bit Data

In addition to receiving standard 8-bit data, the EUSART system supports 9-bit data reception. On the transmit side, the ninth bit is "attached" to the original byte just before the STOP bit. On the receive side, when the RX9 bit of the RCSTA register is set, the ninth data bit will be automatically written to the RX9D bit of the same register. When this byte is received, one should take care of how to read its bits- the RX9D data bit must be read before reading the 8 least significant bits of the RCREG register. Otherwise, the ninth data bit will be automatically cleared.

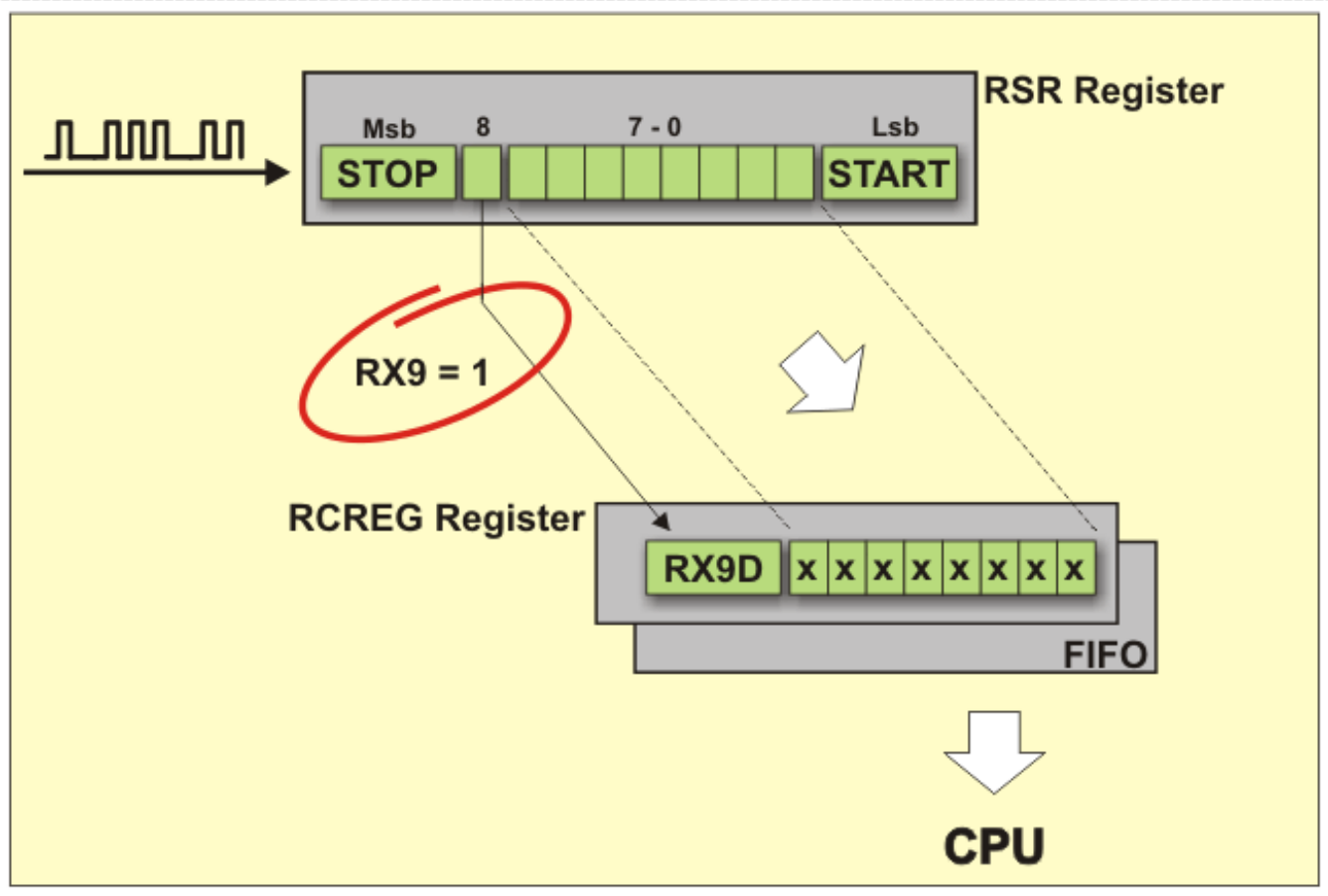
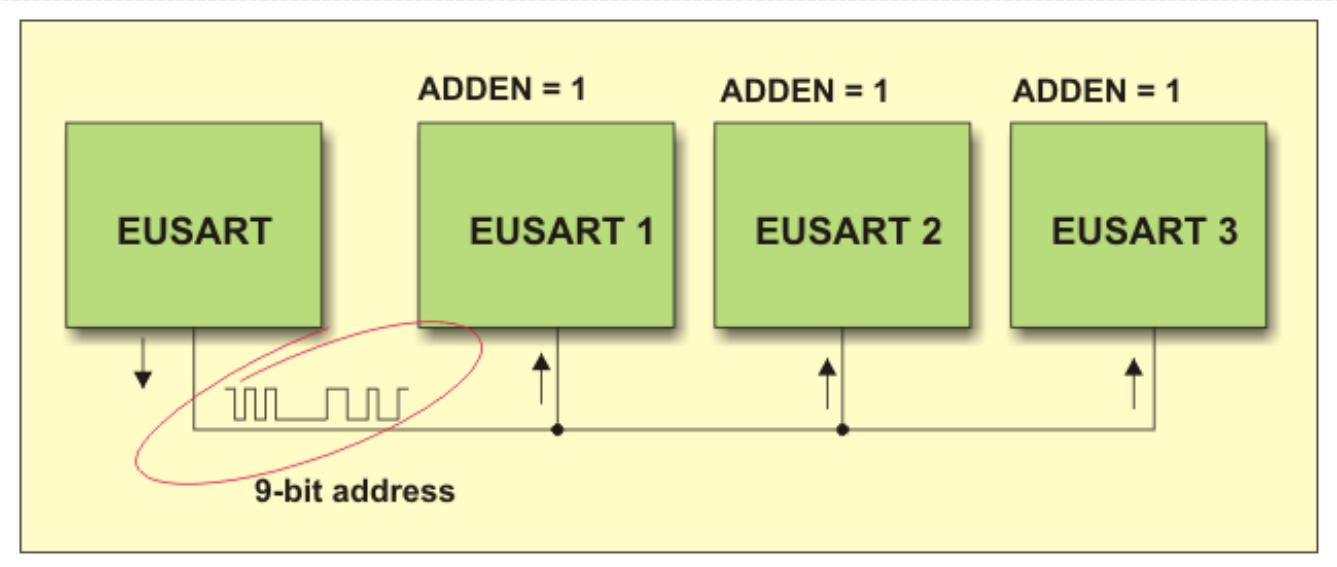


Fig. 6-5 Receiving 9-bit Data

Address Detection

When the ADDEN bit of the RCSTAregister is set, the EUSART module is able to receive only 9-bit data, whereas all 8-bit data will be ignored. Although it seems like a restriction, such modes enable serial communication between several microcontrollers. The principle of operation is simple. The master device sends 9-bit data which represents the address of one microcontroller. All slave microcontrollers sharing the same transmission line, receive this data. Of course, each of them must have the ADDEN bit set because it enables address detection.



Upon receiving this data each slave checks if that address matches its own. Software, in which address match occurs, must disable address detection by clearing its ADDEN bit. The master device keeps on sending 8-bit data. All data passing through the transmission line will be received by "recognized" EUSART module only. Upon receiving the last byte, the slave

device should set the ADDEN bit in order to enable new address detection.

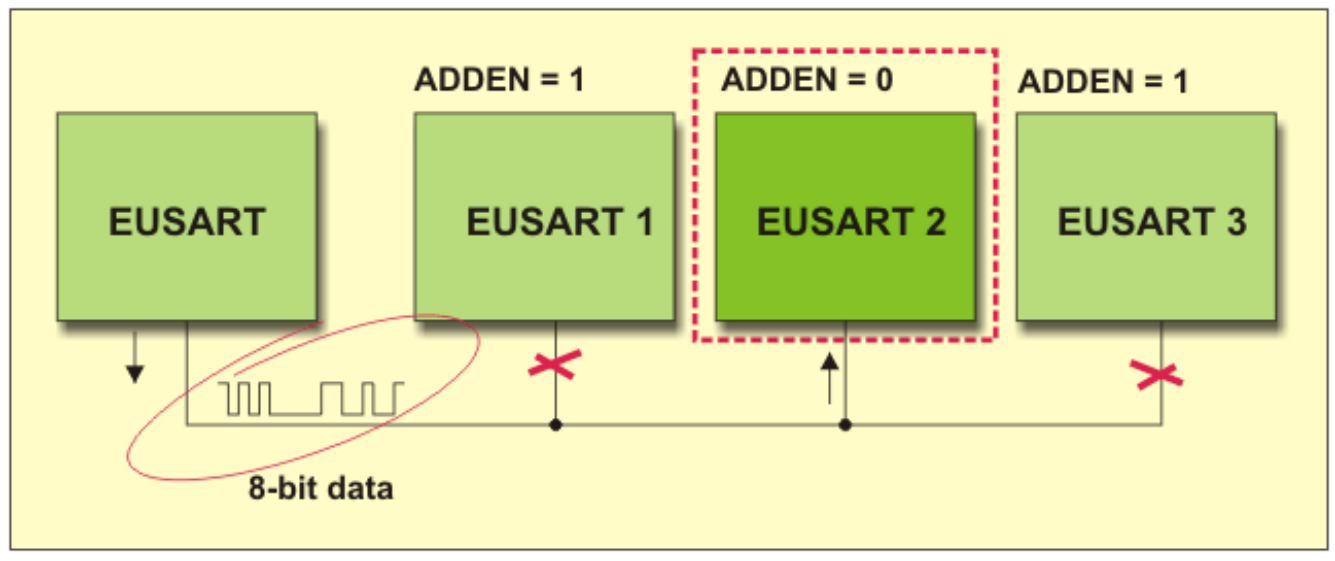


Fig. 6-7 Sending Data

TXSTA Register

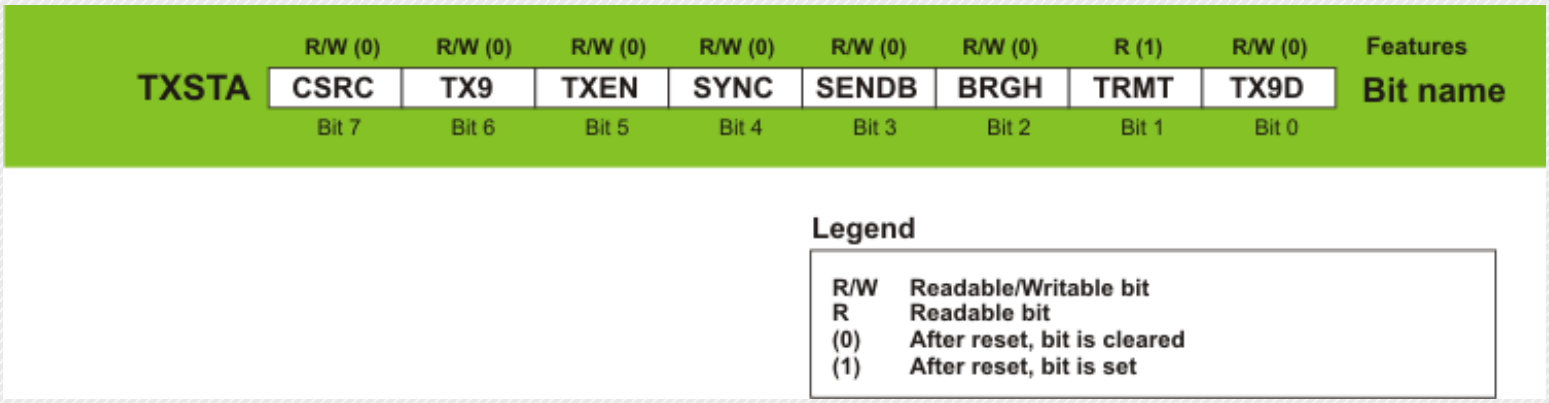


Fig.6-8 TXSTA Register

CSRC - Clock Source Select bit - determines clock source. It is used only in synchronous mode.

- 1 - Master mode. Clock is generated internally from Baud Rate Generator; and
- 0 - Slave mode. Clock is generated from external source.

TX9 - 9-bit Transmit Enable bit

- 1 - 9-bit data transmission via EUSART system; and
- 0 - 8-bit data transmission via EUSART system.

TXEN - Transmit Enable bit

- 1 - Transmission enabled; and
- 0 - Transmission disabled.

SYNC - EUSART Mode Select bit

- 1 - EUSART operates in synchronous mode; and
- 0 - EUSART operates in asynchronous mode.

SENDB - Send Break Character bit is only used in asynchronous mode and only in case it is required to observe LIN bus standard.

- 1 - Sending Break character is enabled; and
- 0 - Break character transmission is completed.

BRGH - High Baud Rate Select bit determines baud rate in asynchronous mode. It does not affect EUSART in synchronous mode.

- 1 - EUSART operates at high speed; and
- 0 - EUSART operates at low speed.

TRMT - Transmit Shift Register Status bit

- 1 - TSR register is empty; and
- 0 - TSR register is full.

TX9D - Ninth bit of Transmit Data can be used as address or parity bit.

RCSTA Register

	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R (0)	R (0)	R (x)	Features
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

R/W	Readable/Writable bit
R	Readable bit
(0)	After reset, bit is cleared
(x)	After reset, bit is unknown

Fig.6-9 RCSTA Register

SPEN - Serial Port Enable bit

- 1 - Serial port enabled. RX/DT and TX/CK pins are automatically configured as input and output respectively; and
- 0 - Serial port disabled.

RX9 - 9-bit Receive Enable bit

- 1 - Receiving 9-bit data via EUSART system; and
- 0 - Receiving 8-bit data via EUSART system.

SREN - Single ReceiveEnable bit is used only in synchronous mode when the microcontroller operates as master.

- 1 - Single receive enabled; and
- 0 - Single receive disable.

CREN - Continuous Receive Enable bit acts differently depending on EUSART mode.

Asynchronous mode:

- 1 - Receiver enabled; and
- 0 - Receiver disabled.

Synchronous mode:

- 1 - Enables continuous receive until the CREN bit is cleared; and
- 0 - Disables continuous receive.

ADDEN - Address Detect Enable bit is only used in address detect mode.

- 1 - Enables address detection on 9-bit data receive; and
- 0 - Disables address detection. The ninth bit can be used as parity bit.

FERR - Framing Error bit

- 1 - On receive, Framing Error is detected; and
- 0 - No framing error.

OERR - Overrun Error bit.

- 1 - On receive, Overrun Error is detected; and
- 0 - No overrun error.

RX9D - Ninth bit of Received Data can be used as address or parity bit.

EUSART Baud Rate Generator (BRG)

If you carefully look at the asynchronous EUSART receiver or transmitter diagram, you will see, in both cases, that clock signal from the local timer BRG is used for synchronization. The same clock source is also used in synchronous mode.

This timer consists of two 8-bit registers comprising one 16-bit register.

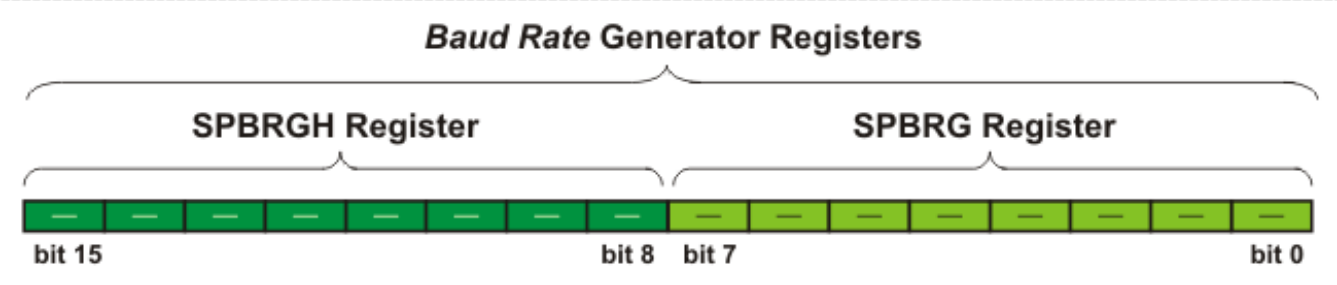


Fig. 6-10 EUSART Baud Rate Generator (BRG)

A number written to these two registers determines the baud rate. Besides, both the BRGH bit of the TXSTA register and the BRGH16 bit of the BAUDCTL register affect clock frequency.

The formula used to determine Baud Rate is given in the table below.

Bits			BRG / EUSART Mode	Baud Rate Formula
SYNC	BRG1G	BRGH		
0	0	0	8-bit / asynchronous	Fosc / [64 (n + 1)]
0	0	1	8-bit / asynchronous	Fosc / [16 (n + 1)]
0	1	0	16-bit / asynchronous	Fosc / [16 (n + 1)]
0	1	1	16-bit / asynchronous	Fosc / [4 (n + 1)]
1	0	X	8-bit / asynchronous	Fosc / [4 (n + 1)]
1	1	X	16-bit / asynchronous	Fosc / [4 (n + 1)]

Table 6-1 Baud Rate

The following tables contain values that should be written to the 16-bit register SPBRG and assigned to the SYNC, BRGH

and BRGH16 bits in order to obtain some of the standard baud rates.

The formulas used to determine the Baud Rate are:

$$\text{Desired Baud Rate} = \frac{Fosc}{64(SPBRGH:SPBRG - 1)}$$

$$SPBRGH:SPBRG = \frac{\frac{Fosc}{\text{Desired Baud Rate}}}{64}$$

$$\text{Error [\%]} = \frac{\text{Calc.Baud Rate - Desired Baud Rate}}{\text{Desired Baud Rate}}$$

Baud Rate	SYNC = 0, BRGH = 0, BRG16 = 0											
	Fosc = 20 MHz			Fosc = 18.432 MHz			Fosc = 11.0592 MHz			Fosc = 11.0592 MHz		
	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)
300	-	-	-	-	-	-	-	-	-	-	-	-
1200	1221	1.73	255	1200	0.00	239	1200	0.00	143	1202	0.16	103
2400	2404	0.16	129	2400	0.00	119	2400	0.00	71	2404	0.16	51
9600	9470	-1.36	32	9600	0.00	29	9600	0.00	17	9615	0.16	12
10417	10417	0.00	29	10286	-1.26	27	10165	-2.42	16	10417	0.00	11
19.2k	19.53	1.73	15	19.2	0.00	14	19.2	0.00	8	-	-	-
57.6k	-	-	-	57.6k	0.00	7	57.6	0.00	2	-	-	-
115.2k	-	-	-	-	-	-	-	-	-	-	-	-

Baud Rate	SYNC = 0, BRGH = 0, BRG16 = 0											
	Fosc = 4 MHz			Fosc = 3.6864 MHz			Fosc = 2 MHz			Fosc = 1 MHz		
	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)
300	300	0.16	207	300	0.00	191	300	0.16	103	300	0.16	51
1200	1202	0.16	51	1200	0.00	47	1202	0.16	25	1202	0.16	12
2400	2404	0.16	25	2400	0.00	23	2404	0.16	12	-	-	-
9600	-	-	-	9600	0.00	5	-	-	-	-	-	-
10417	10417	0.00	5	-	-	-	10417	0.00	2	-	-	-
19.2k	-	-	-	19.2	0.00	2	-	-	-	-	-	-
57.6k	-	-	-	57.6k	0.00	0	-	-	-	-	-	-
115.2k	-	-	-	-	-	-	-	-	-	-	-	-

Baud Rate	SYNC = 0, BRGH = 1, BRG16 = 0											
	Fosc = 20 MHz			Fosc = 18.432 MHz			Fosc = 11.0592 MHz			Fosc = 8 MHz		
	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)
300	-	-	-	-	-	-	-	-	-	-	-	-
1200	-	-	-	-	-	-	-	-	-	-	-	-
2400	-	-	-	-	-	-	-	-	-	2404	0.16	207
9600	9615	0.16	129	9600	0.00	119	9600	0.00	71	9615	0.16	51
10417	10417	0.00	119	10378	-0.37	110	10473	0.53	65	10417	0.00	47
19.2k	19.23k	0.16	64	19.2	0.00	59	19.2k	0.00	35	19231	0.16	25
57.6k	56.82k	-1.36	21	57.6k	0.00	19	57.6k	0.00	11	55556	-3.55	8
115.2k	113.64k	-1.36	10	115.2k	0.00	9	115.2k	0.00	5	-	-	-

Baud Rate	SYNC = 0, BRGH = 1, BRG16 = 0											
	Fosc = 4 MHz			Fosc = 3.6864 MHz			Fosc = 2 MHz			Fosc = 1 MHz		
	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)
300	-	-	-	-	-	-	-	-	-	300	0.16	207
1200	1202	0.16	207	1200	0.00	191	1202	0.16	103	1202	0.16	51
2400	2404	0.16	103	2400	0.00	95	2404	0.16	51	2404	0.16	25
9600	9615	0.16	25	9600	0.00	23	9615	0.16	12	-	-	-
10417	10417	0.00	23	10473	0.00	11	10417	0.00	11	10417	0.00	5
19.2k	19.23k	0.16	12	19.2	0.00	11	-	-	-	-	-	-
57.6k	-	-	-	57.6k	0.00	3	-	-	-	-	-	-
115.2k	-	-	-	115.2k	0.00	1	-	-	-	-	-	-

Baud Rate	SYNC = 0, BRGH = 0, BRG16 = 1											
	Fosc = 20 MHz			Fosc = 18.432 MHz			Fosc = 11.0592 MHz			Fosc = 8 MHz		
	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)
300	300	-0.01	4166	300	0.00	3839	300	0.00	2303	299.9	-0.02	1666
1200	1200	-0.03	1041	1200	0.00	959	1200	0.00	575	1199	-0.08	416
2400	2399	-0.03	520	2400	0.00	479	2400	0.00	287	2404	0.16	207
9600	9615	0.16	129	9600	0.00	119	9600	0.00	71	9615	0.16	51
10417	10417	0.00	119	10378	-0.37	110	10473	0.53	65	10417	0.00	47
19.2k	19.23k	0.16	64	19.2k	0.00	59	19.2k	0.00	35	19.23k	0.16	25
57.6k	56.818	-1.36	21	57.6k	0.00	19	57.6k	0.00	11	55556	-3.55	8
115.2k	113.636	-1.36	10	115.2k	0.00	9	115.2k	0.00	5	-	-	-

Baud Rate	SYNC = 0, BRGH = 0, BRG16 = 1											
	Fosc = 4 MHz			Fosc = 3.6864 MHz			Fosc = 2 MHz			Fosc = 1 MHz		
	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)
300	300.1	0.04	832	300	0.00	767	299.8	-0.108	416	300.5	0.16	207
1200	1202	0.16	207	1200	0.00	191	1202	0.16	103	1202	0.16	51
2400	2404	0.16	103	2400	0.00	95	2404	0.16	51	2404	0.16	25
9600	9615	0.16	25	9600	0.00	23	9615	0.16	12	-	-	-
10417	10417	0.00	23	10473	0.53	21	10417	0.00	11	10417	0.00	5
19.2k	19.23k	0.16	12	19.2k	0.00	11	-	-	-	-	-	-
57.6k	-	-	-	57.6	0.00	3	-	-	-	-	-	-
115.2k	-	-	-	115.2k	0.00	1	-	-	-	-	-	-

Baud Rate	SYNC = 0, BRGH = 1, BRG16 = 1 or SYNC = 1, BRGH16 = 1											
	Fosc = 20 MHz			Fosc = 18.432 MHz			Fosc = 11.0592 MHz			Fosc = 8 MHz		
	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)
300	300	0.00	16665	300	0.00	15359	300	0.00	9215	300	0.00	6666
1200	1200	-0.01	4166	1200	0.00	3839	1200	0.00	2303	1200	-0.02	1666
2400	2400	0.02	2082	2400	0.00	1919	2400	0.00	1151	2401	0.04	832
9600	9597	-0.03	520	9600	0.00	479	9600	0.00	287	9615	0.16	207
10417	10417	0.00	479	10425	0.08	441	10433	0.16	264	10417	0	191
19.2k	19.23k	0.16	259	19.2k	0.00	239	19.2k	0.00	143	19.23k	0.16	103
57.6k	57.47k	-0.22	86	57.6k	0.00	79	57.6k	0.00	47	57.14k	-0.79	34
115.2k	116.3k	0.95	42	115.2k	0.00	39	115.2k	0.00	23	117.6k	2.12	16

Baud Rate	SYNC = 0, BRGH = 1, BRG16 = 1 or SYNC = 1, BRGH16 = 1											
	Fosc = 4 MHz			Fosc = 3.6864 MHz			Fosc = 2 MHz			Fosc = 1 MHz		
	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)	Actual Rate	Error %	SPBRG value (dec.)
300	300	0.01	3332	300	0.00	3071	299.9	-0.02	1666	300.1	0.04	832
1200	1200	0.04	832	1200	0.00	767	1199	-0.08	416	1202	0.16	207
2400	2398	0.08	416	2400	0.00	383	2404	0.16	207	2404	0.16	103
9600	9615	0.16	103	9600	0.00	96	9615	0.16	51	9615	0.16	25
10417	10417	0.00	95	10473	0.53	87	10417	0.00	47	10417	0.00	23
19.2k	19.23k	0.16	51	19.2k	0.00	47	19.23k	0.16	25	19.23k	0.16	12
57.6k	58.82k	2.12	16	57.6k	0.00	15	55.56k	-3.55	8	-	-	-
115.2k	111.1k	-3.55	8	115.2k	0.00	7	-	-	-	-	-	-

Table 6-2 Determining Baud Rate

BAUDCTL Register

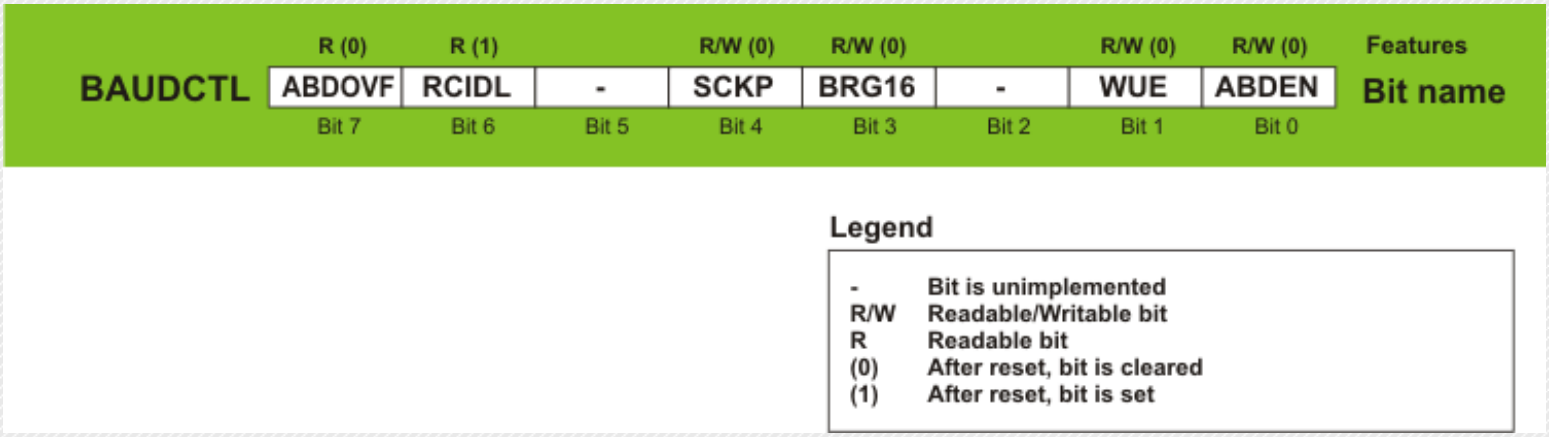


Fig. 6-11 BAUDCTL Register

ABDOVF - Auto-Baud Detect Overflow bit is only used in asynchronous mode during baud rate detection.

- 1 - Auto-baud timer overflowed; and
- 0 - Auto-baud timer did not overflow.

RCIDL - Receive Idle Flag bit is only used in asynchronous mode.

- 1 - Receiver is idle; and
- 0 - START bit has been received and receiving is in progress.

SCKP - Synchronous Clock Polarity Select bit acts differently depending on EUSART mode.

Asynchronous mode:

- 1 - Transmit inverted data to the RC6/TX/CK pin; and
- 0 - Transmit non-inverted data to the same pin.

Synchronous mode:

- 1 - Synchronization on rising edge of the clock; and
- 0 - Synchronization on falling edge of the clock.

WUE Wake-up Enable bit

- 1 - Receiver waits for a falling edge on the RC7/RX/DT pin to start waking up the microcontroller from sleep mode; and
- 0 - Receiver operates normally.

ABDEN - Auto-Baud Detect Enable bit is used in asynchronous mode only.

- 1 - Auto-baud detect mode is enabled. Bit is automatically cleared on baud rate detect; and
- 0 - Auto-baud detect mode is disabled.

In Short:

Sending data via asynchronous EUSART communication:

1. The desired baud rate should be set by using bits BRGH (TXSTA register) and BRG16 (BAUDCTL register) and registers SPBRGH and SPBRG;
2. The SYNC bit (TXSTA register) should be cleared and the SPEN bit should be set (RCSTA register) in order to enable serial port;
3. On 9-bit data transmission, the TX9 bit of the TXSTA register should be set;
4. Data transmission is enabled by setting bit TXEN of the TXSTA register. Bit TXIF of the PIR1 register is automatically set;

5. If needed the bit TXEN causes an interrupt, the GIE and PEIE bits of the INTCON register should be set;
6. On 9-bit data transmission, value of the ninth bit should be written to the TX9D bit of the TXSTA register; and
7. Transmission starts by writing 8-bit data to the TXREG register.

Receiving data via asynchronous EUSART communication:

1. *Baud Rate* should be set by using bits BRGH (TXSTA register) and BRG16 (BAUDCTL register) and registers SPBRGH and SPBRG;
2. The SYNC bit (TXSTA register) should be cleared and the SPEN bit should be set (RCSTA register) in order to enable serial port;
3. If it is necessary the data receive causes an interrupt, both the RCIE bit of the PIE1 register and bits GIE and PEIE of the INTCON register should be set;
4. On 9-bit data receive, the RX9 bit of the RCSTA register should be set;
5. Data receive should be enabled by setting the CREN bit of the RCSTA register;
6. The RCSTA register should be read to get information on possible errors which have occurred during transmission. On 9-bit data receive, the ninth bit will be stored in this register; and
7. Received 8-bit data stored in the RCREG register should be read.

Setting Address Detection Mode:

1. Baud Rate should be set by using bits BRGH (TXSTA register) and BRG16 (BAUDCTL register) and registers SPBRGH and SPBRG;
2. The SYNC bit (TXSTA register) should be cleared and the SPEN bit should be set (RCSTA register) in order to enable serial port;
3. If it is necessary the data receive causes an interrupt, the RCIE bit of the PIE1 bit as well as bits GIE and PEIE of the INTCON register should be set;
4. The RX9 bit of the RCSTA register should be set;
5. The ADDEN of the RCSTA register should be set, which enables a data to be interpreted as address;
6. Data receive is enabled by setting the CREN bit of the RCSTA register;
7. Immediately upon 9-bit data is received, the RCIF bit of the PIR1 register will be automatically set. If enabled, an interrupt occurs;
8. The RCSTA register should be read in order to get information on possible errors which have occurred during transmission. The ninth bit RX9D is always set; and
9. Received 8-bits stored in the RCREG register should be read. It should be checked whether the combination of these bits matches the predefined address. If the match occurs, it is necessary to clear the ADDEN bit of the RCSTA register, which enables further 8-bit data receive.

Master Synchronous Serial Port Module

MSSP module (*Master Synchronous Serial Port*) is a very useful, but at the same time one of the most complex circuit within the microcontroller. It enables high speed communication between a microcontroller and other peripherals or microcontroller devices by using few input/output lines (maximum two or three). Therefore, it is commonly used to connect the microcontroller to LCD displays, A/D converters, serial EEPROMs, shift registers etc. The main feature of this type of communication is that it is synchronous and suitable for use in systems with a single master and one or more slaves. A master device contains the necessary circuitry for baud rate generation and supplies the clock for all devices in the system. Slave devices may in that way eliminate the internal clock generation circuitry. The MSSP module can operate in one of two modes:

- SPI mode (Serial Peripheral Interface)
- I²C mode (Inter-Integrated Circuit)

As seen in figure 6-12 below, one MSSP module represents only a half of the hardware needed to establish serial communication, while another half is stored in the device the data is exchanged with. Even though the modules on both ends of the line are the same, their modes are essentially different depending on whether they operate as a **Master** or a **Slave**:

If the microcontroller to be programmed controls another device or circuit (peripherals), it should operate as a master device. A module defined as such will generate clock when needed, i.e. only when data receive and transmit is required by the software. It depends on the master whether the connection will be established or not. Otherwise, if the microcontroller to be programmed is a part of some peripheral which belongs to some more complex device (for example PC), then it should operate as a slave device. As such, it always has to wait for request for data transfer from master device.

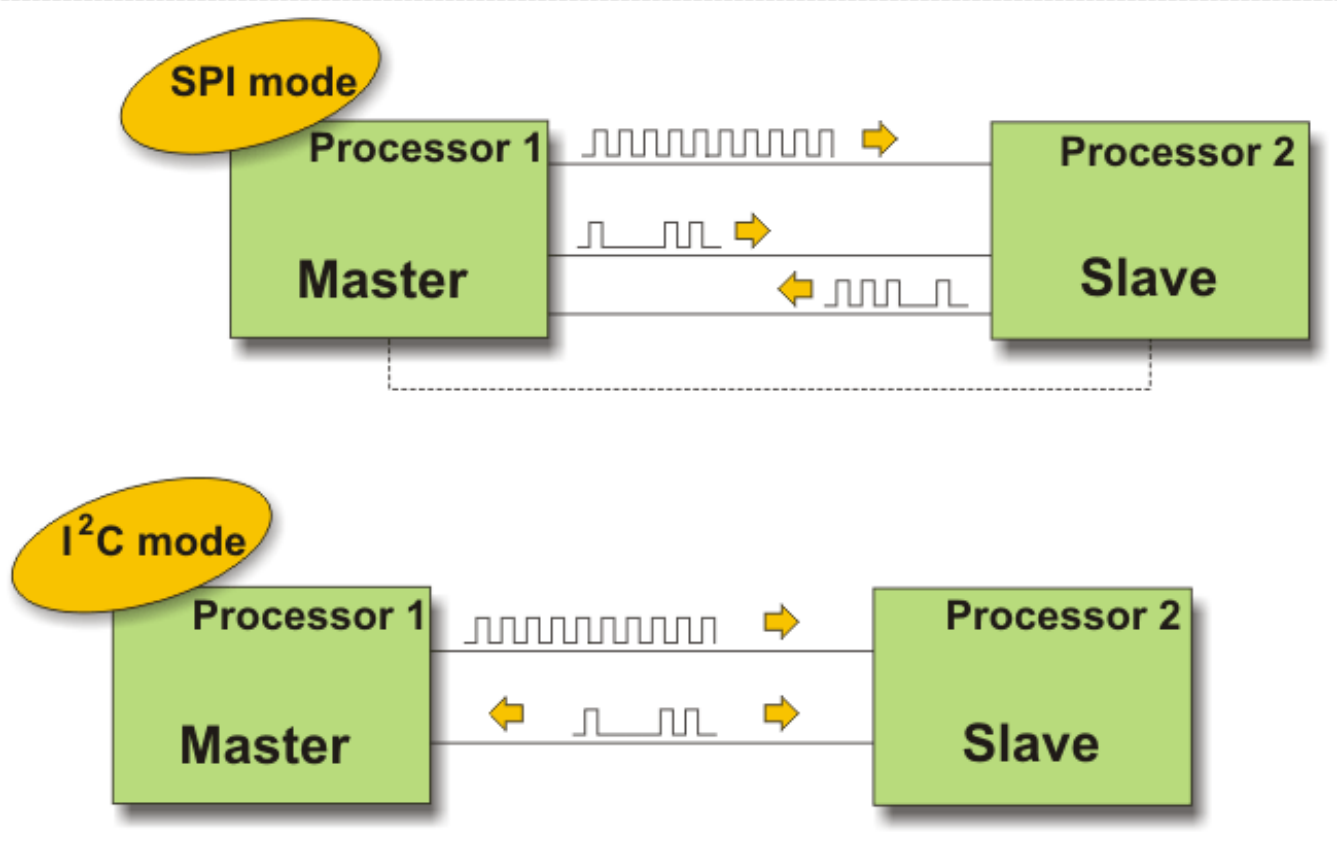


Fig.6-12 MSSP Module

SPI Mode

The SPI mode allows 8 bits of data to be transmitted and received simultaneously using 3 input/output lines:

- **SDO** - Serial Data Out - transmit line;
- **SDI** - Serial Data In - receive line; and
- **SCK** - Serial Clock - synchronization line.

In addition to these three lines, if the microcontroller exchanges data with several peripheral devices, the forth line (SS) may be also used. Refer to figure 6-13 below.

SS - Slave Select - is additional pin used for specific device selection. It is active only in case the microcontroller is in slave mode, i.e. when the external - master device requires data exchange.

When operating in SPI mode, MSSP module uses in total of 4 registers:

- SSPSTAT status register;
- SSPCON control register;
- SSPBUF buffer register; and
- SSPSR shift register (not directly available)

The first three registers are writable/readable and can be changed at any moment, while the forth register, since not available, is used for converting data into "serial" format.

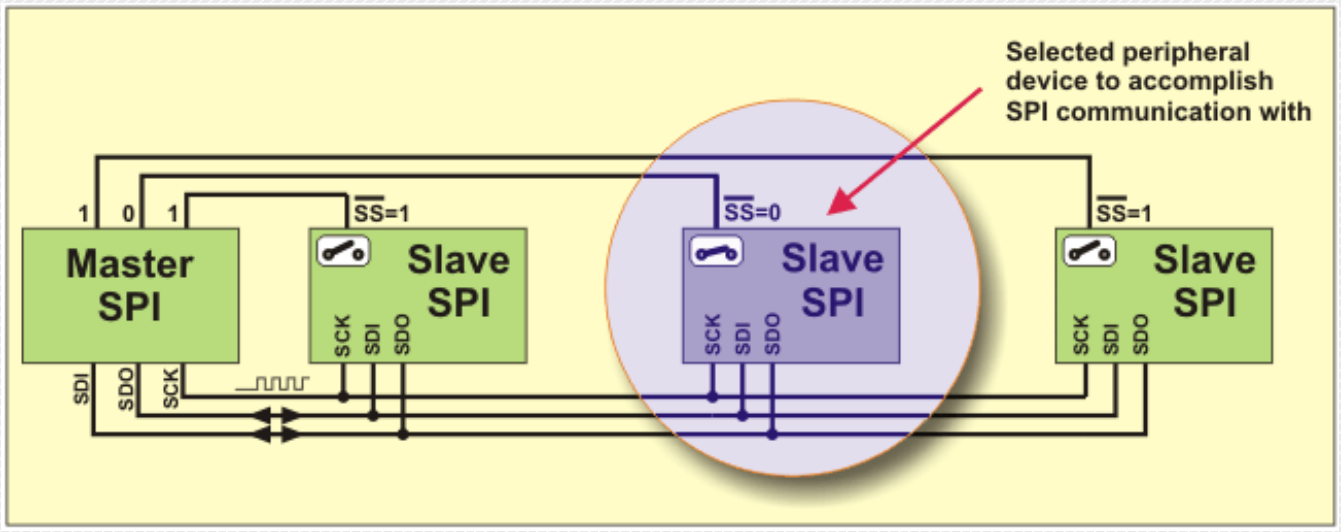


Fig. 6-13 SPI Mode

As seen in figure 6-14, the central part of the SPI module consists of two registers connected to pins for receive, transmit and synchronization.

Shift register (SSPRS) is directly connected to the microcontroller pins and used for data transmission in serial format. The SSPRS register has its input and output and shifts the data in and out of device. In other words, each bit appearing on input (receive line) simultaneously shifts another bit toward output (transmit line).

The **SSPBUF** register (Buffer) is a part of memory used to temporarily hold the data written to the SSPRS until the received data is ready. Upon receiving all 8 bits of data, that byte is moved to the SSPBUF register. This double buffering of the received data (SSPBUF) allows the next byte to start reception before reading the data that was just received. Any write to the SSPBUF register during transmission/reception of data will be ignored. Since having been the most accessed, this register is considered the most important from the programmers' point of view.

Namely, if mode settings are neglected, data transfer via SPI actually means to write and read data from this register, while another "acrobatics" such as moving registers are automatically performed by hardware.

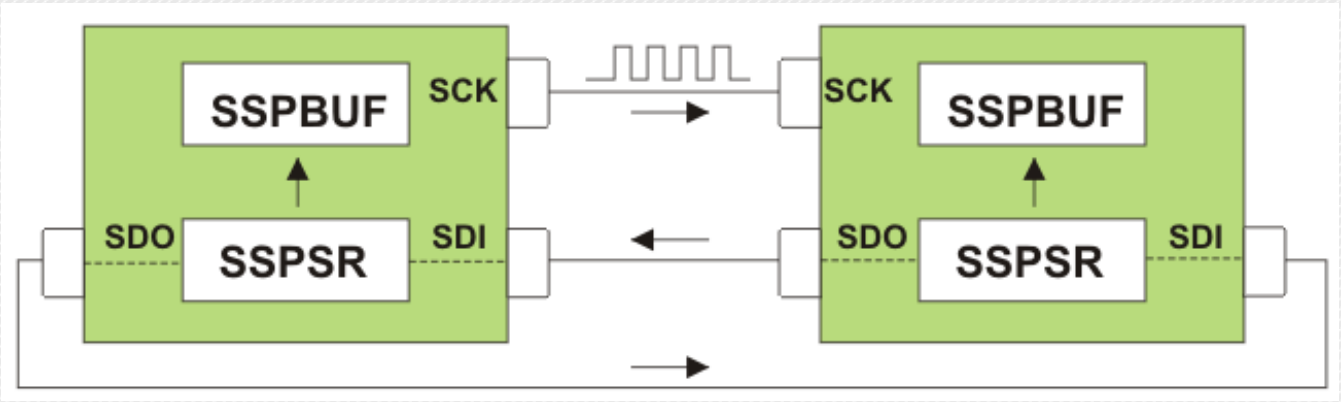


Fig. 6-14 SPI Mode

In short:

Prior to initializing the SPI, it is necessary to specify several options:

- Master mode (SCK pin is the clock output);
- Slave mode (SCK pin is the clock input);
- Data input phase- middle or end of data output time (SMP bit);
- Clock edge (CKE bit);
- Baud Rate (only in Master mode); and
- Slave select mode (Slave mode only).

Step 1.
Data to transmit should be written to the buffer register SSPBUF. Immediately after that, if the SPI module operates in master mode, the microcontroller will automatically perform the following steps 2, 3 and 4. If the SPI module operates as Slave, the microcontroller will not perform these steps until the SCK pin detects clock signal.

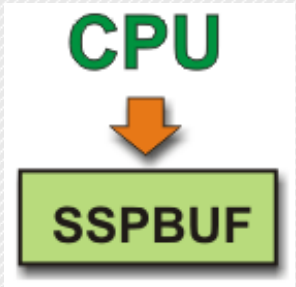


Fig. 6-15 Step 1

Step 2.
This data is now moved to the SSPSR register and the SSPBUF register is not cleared.

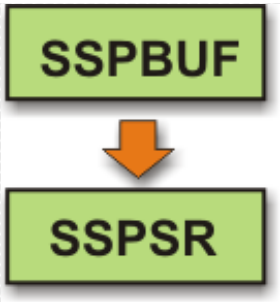


Fig. 6-16 Step 2

Step 3.
Synchronized with clock signal, this data is shifted to the output pin (MSB bit first) while the register is simultaneously being filled with bits through input pin. In Master mode, the microcontroller itself generates clock, while the Slave mode uses external clock (pin SCK).

Fig. 6-17 Step 3

Step 4.
The SSPSR register is full once the 8 bits of data have been received. It is indicated by setting the BF and SSPIF bits. The received data (that byte) is automatically moved from the SSPSR register to the SSPBUF register. Since data transfer via serial communication is performed automatically, the rest of the program is normally executed while data transfer is in progress. In that case, the function of the SSPIF bit is to generate interrupt when one byte transmission is completed.

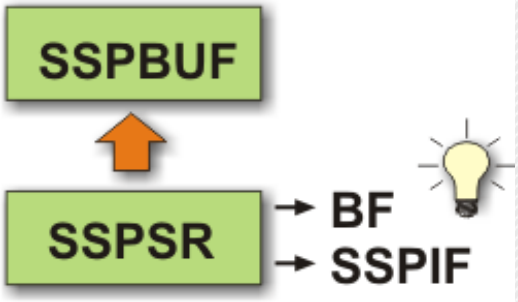


Fig. 6-18 Step 4

Step 5.
At last, the data stored in the SSPBUF register is ready for use and moved to any register available.

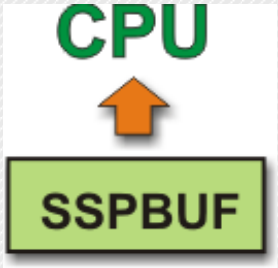


Fig. 6-19 Step 5

I²C mode

I²C mode (*Inter IC Bus*) is especially suitable when the microcontroller and integrated circuit, which the microcontroller should exchange data with, are within the same device. It is commonly about another microcontrollers or specialized, cheap integrated circuits belonging to the new generation of so called "smart peripheral components" (memories, temperature sensors, real-time clocks etc.)

Similar to serial communication in SPI mode, data transfer in I²C mode is synchronous and bidirectional. This time only two pins are used for data transfer. These are the SDA (Serial Data) and SCL (Serial Clock) pins. The user must configure these pins as inputs or outputs through the TRISC bits.

Perhaps it is not directly visible. By observing particular rules (protocols), this mode enables up to 122 different components to be simultaneously connected in a simple way by using only two valuable I/O pins. Briefly, everything works as follows: Clock necessary to synchronize the operation of both devices is always generated by the master device (microcontroller) and its frequency directly affects baud rate. There are protocols allowing maximum 3,4 MHz clock

frequency (so called high-speed I²C bus), but the clock frequency of the most frequently used protocol is limited to 100 KHz. There is no limit in case of minimal frequency.

When **master** and **slave** components are synchronized by the clock, every data exchange is always initialized by master. Once the MSSP module has been enabled, it waits for a Start condition to occur. First the master device sends the START bit (logic zero) through the SDA pin, then the 7-bit address of the selected slave device, and finally, the bit which requires data write (0) or read (1) to that device. Accordingly, following the start condition, the eight bits are shifted into the SSPSR register. All slave devices share the same transmission line and all will simultaneously receive the first byte, but only one of them has the address to match.

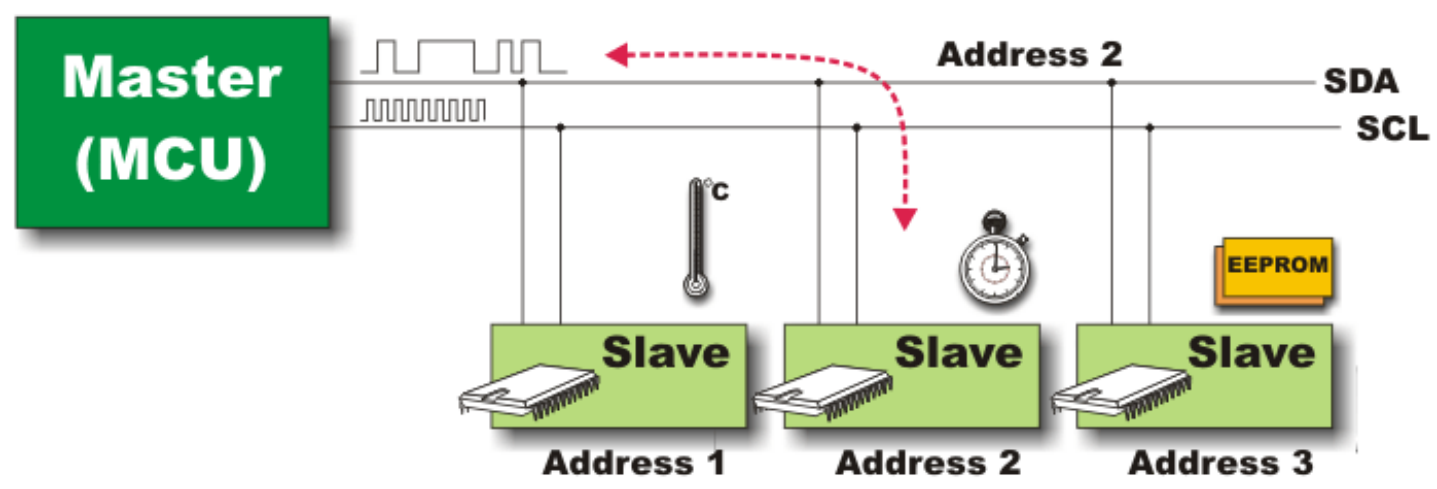


Fig. 6-20 Master and Slave Configuration

Once the first byte has been sent (only 8-bit data are transmitted), master goes into receive mode and waits for acknowledgment from the receive device that address match has occurred. If the slave device sends acknowledge data bit (1), data transfer will be continued until the master device (microcontroller) sends the Stop bit.

This is the simplest explanation of how two components communicate. If needed, this microcontroller is able to control more complicated situations when 1024 different components, shared by several different master devices, are connected. Such devices are rarely used in practice and there is no need to discuss them at greater length.

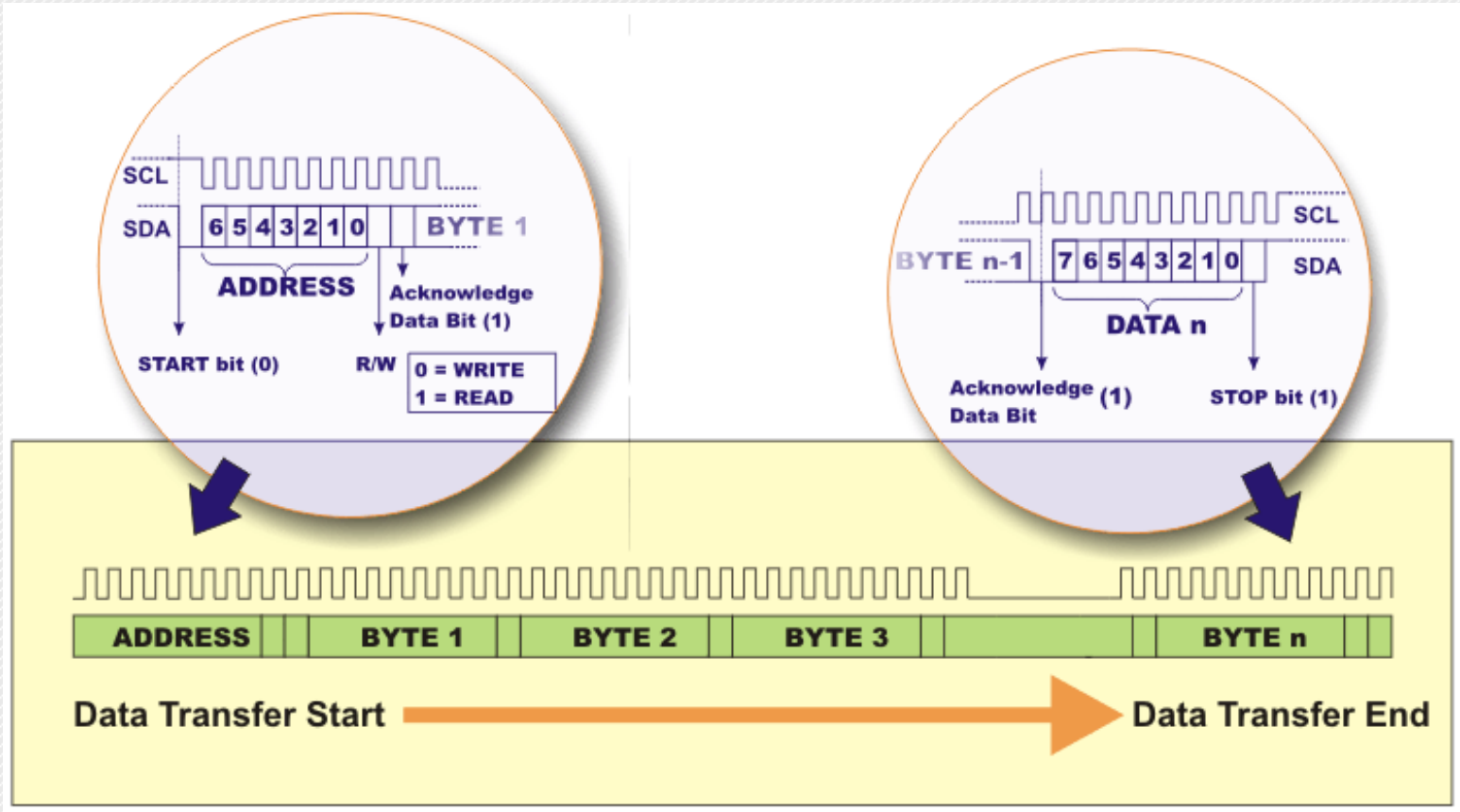
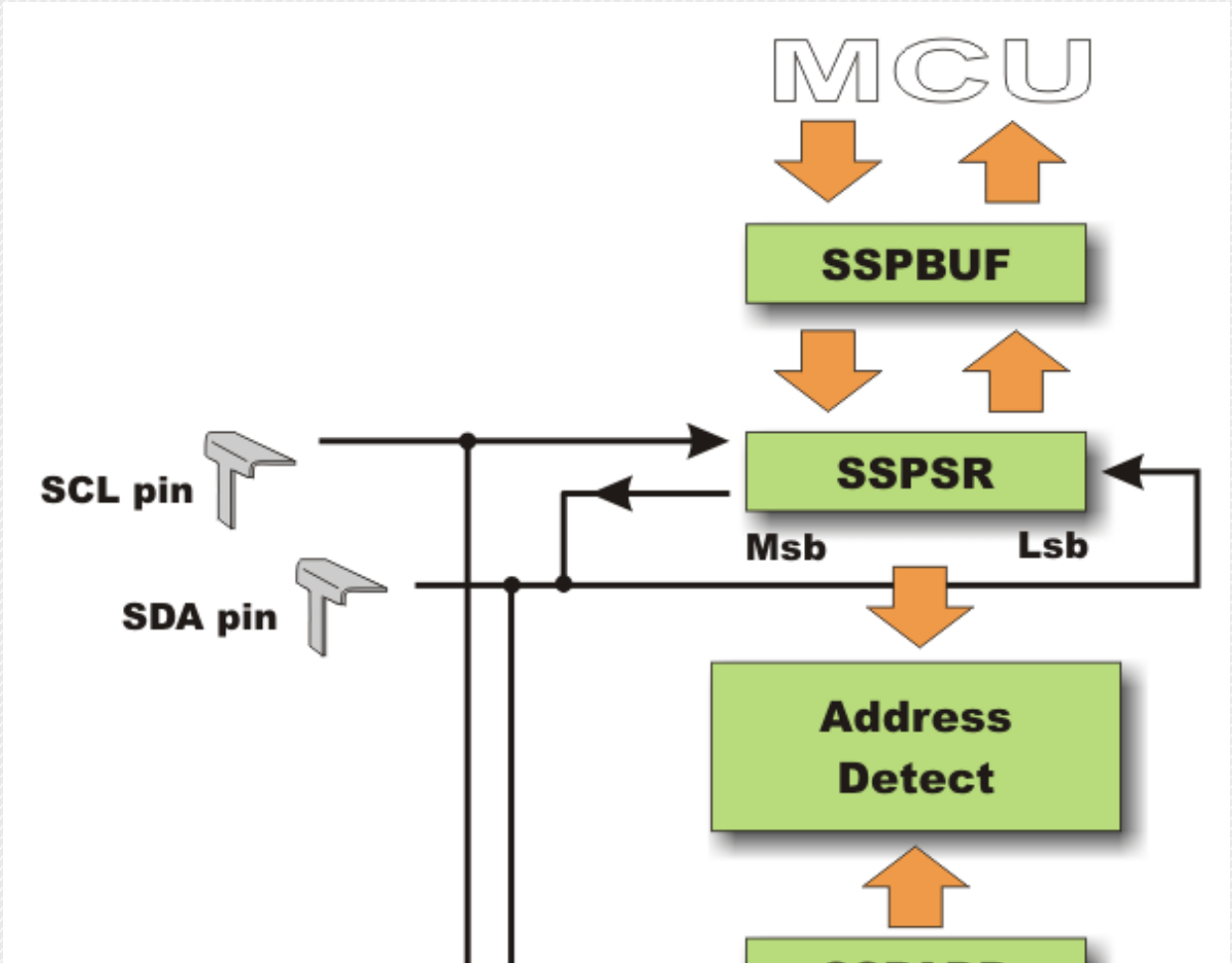


Fig. 6-21 Data Transfer

Figure below shows the block diagram of the MSSP module in I²C mode.



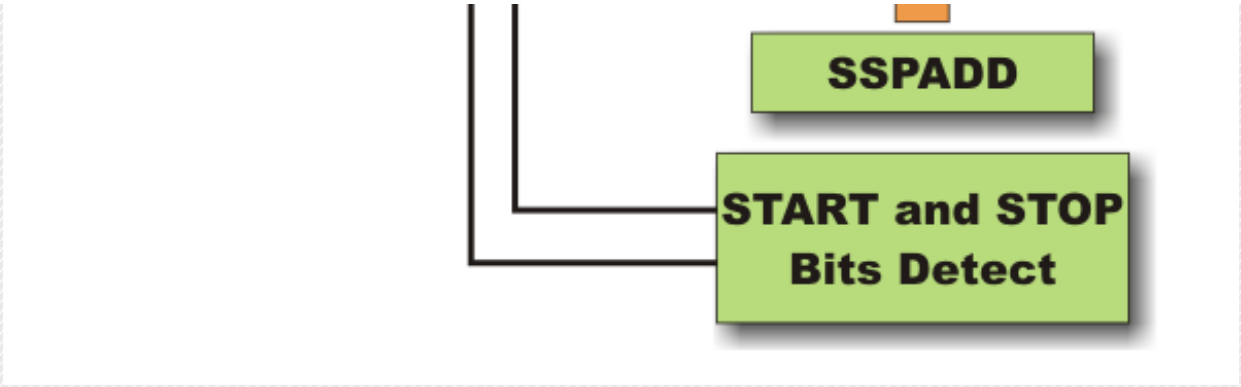


Fig. 6-22 MSSP Block Diagram in I²C Mode

The MSSP module uses six registers for I²C operation. Some of them are shown in figure above:

- SSPCON;
- SSPCON2;
- SSPSTAT;
- SSPBUF;
- SSPSR; and
- SSPADD.

SSPSTAT Register

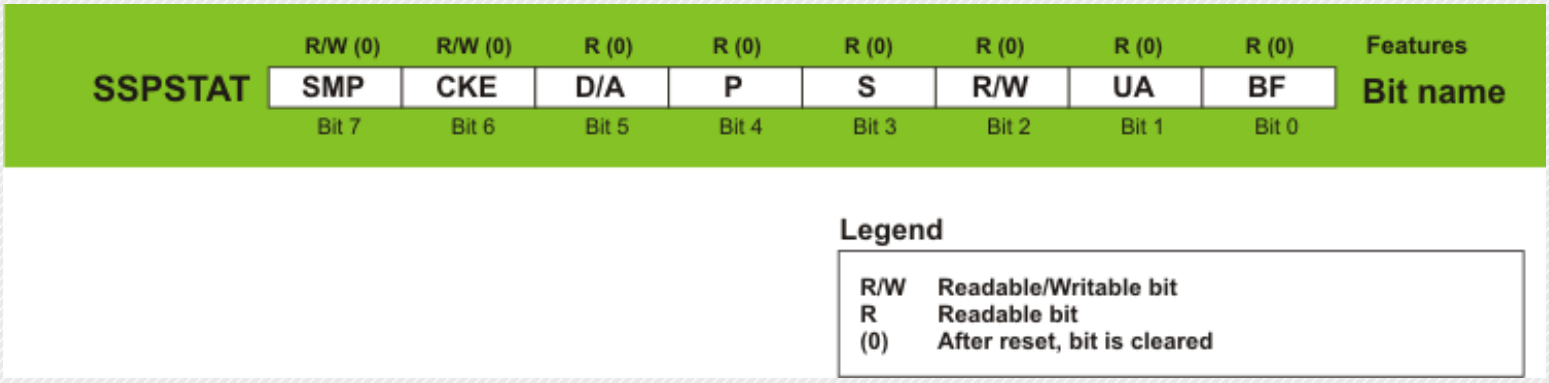


Fig. 6-23 SSPSTAT Register

SMP Sample bit

SPI master mode - This bit determines input data phase.

- 1 - Logic state is read at end of data output time; and
- 0 - Logic state is read in the middle of data output time.

SPI slave mode This bit must be cleared when SPI is used in Slave mode.

I²C mode (master or slave)

- 1 - Slew rate control disabled for standard speed mode (100kHz); and
- 0 - Slew rate control enabled for high speed mode (400kHz).

CKE - Clock Edge Select bit selects synchronization mode.

CKP = 0:

- 1 - Data is transmitted on rising edge of clock pulse (0 - 1); and
- 0 - Data is transmitted on falling edge of clock pulse (1 - 0).

CKP = 1:

- 1 - Data is transmitted on falling edge of clock pulse (1 - 0); and
- 0 - Data is transmitted on rising edge of clock pulse (0 - 1).

D/A - Data/Address bit is used in I²C mode only.

- 1 - Indicates that the last byte received or transmitted was data; and
- 0 - Indicates that the last byte received or transmitted was address.

P - Stop bit is used in I²C mode only.

- 1 - STOP bit was detected last; and
- 0 - STOP bit was not detected last.

S - Start bit is used in I²C mode only.

- 1 - START bit was detected last; and
- 0 - START bit was not detected last.

R/W - Read Write bit is used in I²C mode only. This bit holds the R/W bit information following the last address match. This bit is only valid from the address match to the next Start bit, Stop bit or not ACK bit.

In I²C slave mode
- 1 - Data read; and
- 0 - Data write.

In I²C master mode
- 1 - Transmit is in progress; and
- 0 - Transmit is not in progress.

UA - Update Address bit is used in 10-bit I²C mode only.

- 1 - Indicates that it is necessary to update the address in the SSPADD register; and
- 0 - Address in the SSPADD register is correct and does not need to be updated.

BF Buffer Full Status bit

During data receive (in SPI and I²C modes)
- 1 - Receive complete. The SSPBUF register is full; and
- 0 - Receive not complete. The SSPBUF register is empty.

During data transmit (in I²C mode only)
- 1 - Data transmit in progress (does not include the bits ACK and STOP); and
- 0 - Data transmit complete (does not include the bits ACK and STOP).

SSPCON Register

	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	R/W (0)	Features
SSPCON	WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

R/W	Readable/Writable bit
(0)	After reset, bit is cleared

Fig. 6-24 SSPCON Register

WCOL Write Collision Detect bit

- 1 - Collision detected. A write to the SSPBUF register was attempted while the I²C conditions were not valid for a transmission to start; and
- 0 - No collision.

SSPOV Receive Overflow Indicator bit

- 1 - A new byte is received while the SSPSR register still holds the previous data. Since there is no space for new data receive, one of these two bytes must be cleared. In this case, data in SSPSR is lost; and
- 0 - Serial data is correctly received.

SSPEN - Synchronous Serial Port Enable bit determines the microcontroller pins function and initializes MSSP module:

In SPI mode
- 1 - Enables MSSP module and configures pins SCK, SDO, SDI and SS as the source of the serial port pins; and
- 0 - Disables MSSP module and configures these pins as I/O port pins.

In I²C mode
- 1 - Enables MSSP module and configures pins SDA and SCL as the source of the serial port pins; and
- 0 - Disables MSSP module and configures these pins as I/O port pins.

CKP - Clock Polarity Select bit is not used in I²C master mode.

In SPI mode
- 1 - Idle state for clock is a high level; and
- 0 - Idle state for clock is a low level.

In I²C slave mode
- 1 - Enables clock; and
- 0 - Holds clock low. Used to provide more time for data stabilization.

SSPM3-SSPM0 - Synchronous Serial Port Mode Select bits. SSP mode is determined by combining these bits:

SSPM3	SSPM2	SSPM1	SSPM0	Mode
0	0	0	0	SPI master mode, clock = Fosc/4
0	0	0	1	SPI master mode, clock = Fosc/16
0	0	1	0	SPI master mode, clock = Fosc/64
0	0	1	1	SPI master mode, clock = (output TMR)/2
0	1	0	0	SPI slave mode, SS pin control enabled
0	1	0	1	SPI slave mode, SS pin control disabled, SS can be used as I/O pin
0	1	1	0	I²C slave mode, 7-bit address used
0	1	1	1	I²C slave mode, 10-bit address used
1	0	0	0	I²C master mode, clock = Fosc / [4(SSPAD+1)]
1	0	0	1	Mask used in I²C slave mode
1	0	1	0	Not used
1	0	1	1	I²C controlled master mode
1	1	0	0	Not used
1	1	0	1	Not used
1	1	1	0	I²C slave mode, 7-bit address used,START and STOP bits enable interrupt
1	1	1	1	I²C slave mode, 10-bit address used,START and STOP bits enable interrupt

Table 6-3 Synchronous Serial Port Mode Select Bits

SSPCON2 Register

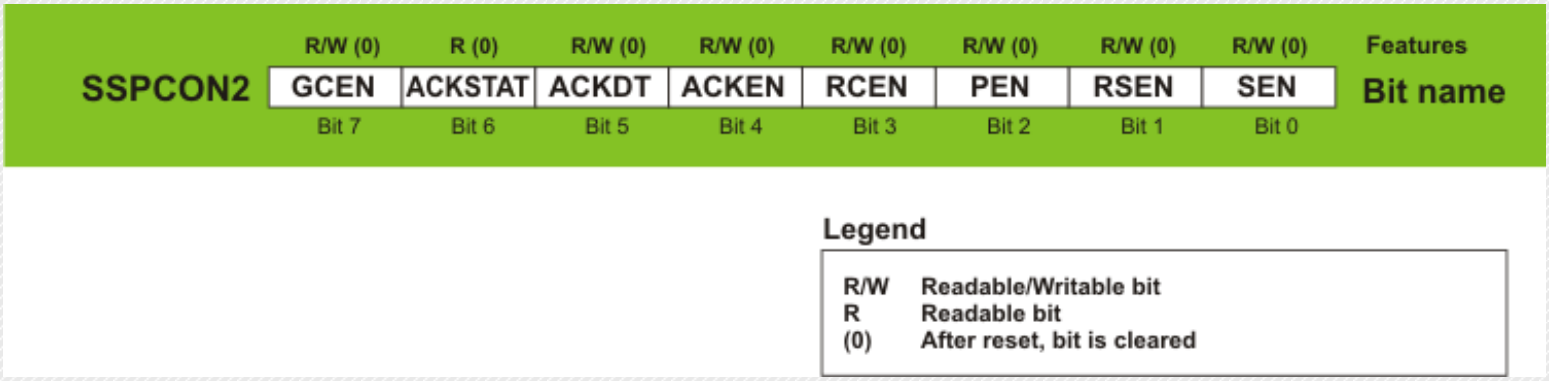


Fig. 6-25 SSPCON2 Register

GCEN - General Call Enable bit

In I²C slave mode only
- 1 - Enables interrupt when a general call address (0000h) is received in the SSPSR; and
- 0 - General call address disabled.

ACKSTAT - Acknowledge Status bit

In I²C Master Transmit mode only
- 1 - Acknowledge was not received from slave; and
- 0 - Acknowledge was received from slave.

ACKDT - Acknowledge data bit

In I²C Master Receive mode only
- 1 - Not Acknowledge; and
- 0 - Acknowledge.

ACKEN - Acknowledge condition Enable bit

In I²C Master Receive mode
- 1 - Initiate acknowledge condition on SDA and SCL pins and transmit ACKDT data bit. It is automatically cleared by hardware; and
- 0 - Acknowledge condition is not initiated.

RCEN - Receive Enable bit

In I²C Master mode only
- 1 - Enables data receive in I²C mode; and
- 0 - Receive disabled.

PEN - STOP condition Enable bit

In I²C Master mode only
- 1 - Initiates STOP condition on pins SDA and SCL. Afterwards, this bit is automatically cleared by hardware; and
- 0 - STOP condition is not initiated.

RSEN - Repeated START Condition Enabled bit

In I²C master mode only
- 1 - Initiates START condition on pins SDA and SCL. Afterwards, this bit is automatically cleared by hardware; and
- 0 - Repeated START condition is not initiated.

SEN - START Condition Enabled/Stretch Enabled bit

In I²C Master mode only
- 1 - Initiate START condition on pins SDA and SCL. Afterwards, this bit is automatically cleared by hardware; and
- 0 - START condition is not initiated.

I²C in Master Mode

The most common case is when the microcontroller operates as a master and the peripheral component as a slave. This is why this book covers just this mode. It is also considered that the address consists of 7 bits and device contains only one microcontroller (one master device).

In order to enable MSSP module in this mode, it is necessary to do the following:

Set baud rate (SSPADD register), turn off slew rate control (by setting the SMP bit of the SSPSTAT register) and select master mode (SSPCON register). After the preparation has been finished and module has been enabled (SSPCON register: SSPEN bit), one should wait for internal electronics to signal that everything is ready for data transmission, i.e. the SSPIF bit of the PIR1 register is set.

This bit should be cleared by software and after that, the microcontroller is ready to start "communication" with peripherals.

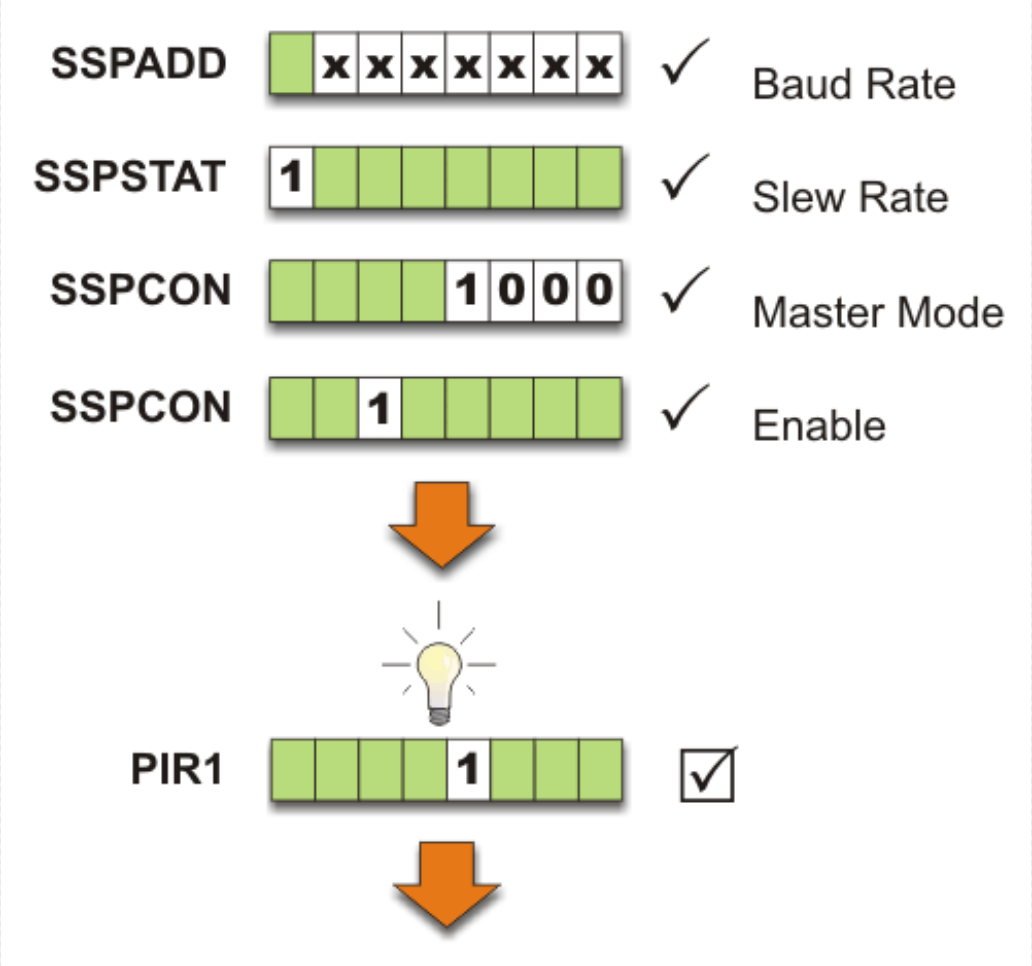


Fig. 6-27 I²C in Master Mode

Data Transmission in I²C Master Mode

Each clock condition on the SDA pin starts with logic zero (0) which appears upon setting the SEN bit of the SSPCON2 register. Even enabled, the microcontroller has to wait a certain time before it starts communication. It is the so called "Start condition" during which internal preparations and checks are performed. If all conditions are met, the SSPIF bit of the PIR1 is set and data transfer starts as soon as the SSPBUF register is loaded.

Since maximum 112 integrated circuits may simultaneously share the same transmission line, the first data byte must contain address which matches only one slave device. Each component has its own address listed in the proper data sheet. The eighth bit of the first data byte specifies direction of data transmission, the microcontroller is to send or receive data. In this case, it is all about data receive and the eighth bit therefore is logic zero (0).

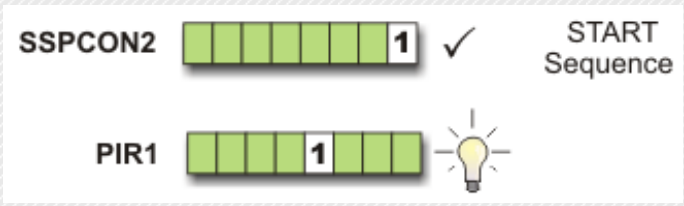
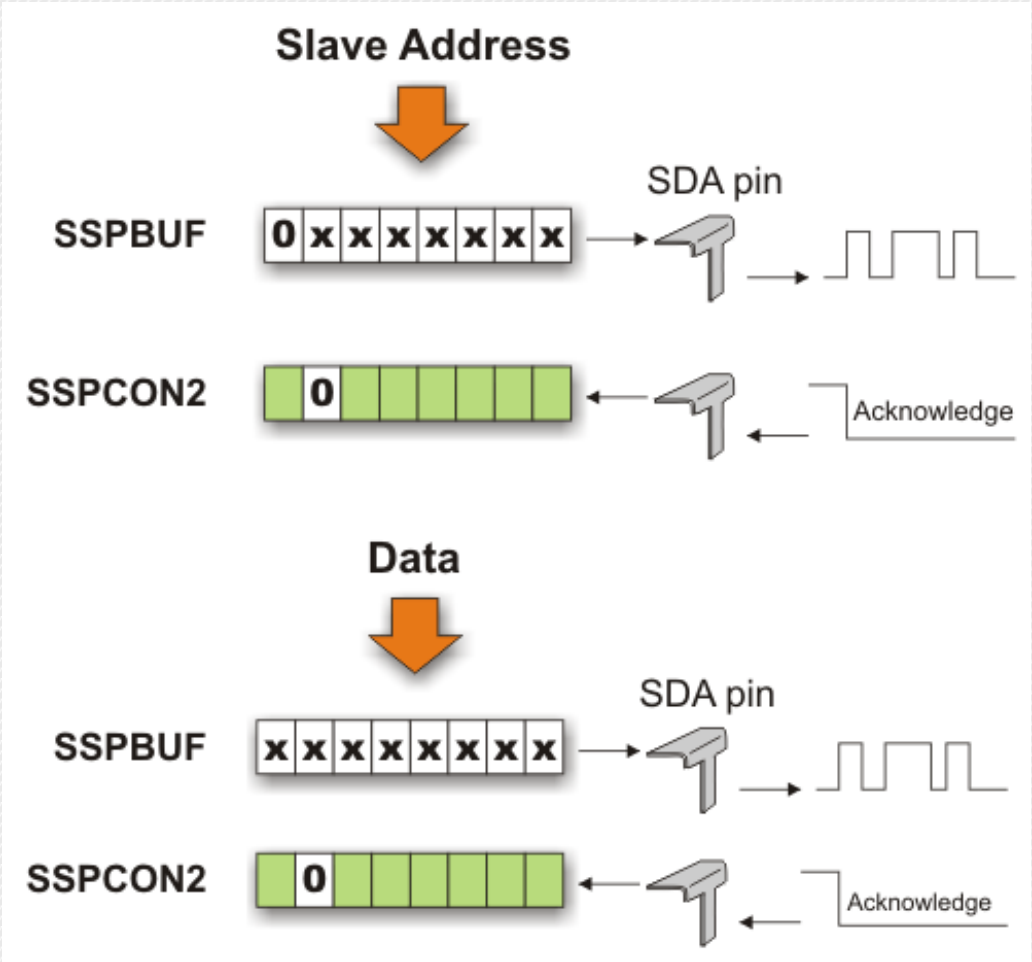


Fig. 6-28 Data Transmission in I²C Master Mode

When address match occurs, the microcontroller has to wait for the acknowledge data bit. The slave device acknowledges address match by clearing the ASKSTAT bit of the SSPCON2 register. If the match properly occurred, all bytes representing data are transmitted in the same way.

Data transmission ends by setting the SEN bit of the SSPCON2 register. The so called STOP condition occurs, which enables the SDA pin to receive pulse condition: Start - Address - Acknowledge - Data - Acknowledge ....Data - Acknowledge - Stop!

Fig.6-29 Data Transmission in I²C Master Mode



Data Reception in I²C Master Mode

Preparations for data reception are similar to those for data transmission, with exception that the last bit of the first sent byte (containing address) is logic one (1). It specifies that master expects to receive data from addressed slave device. With regard to the microcontroller, the following events occur:

After internal preparations are finished and START bit is set, slave device starts sending one byte at a time. These bytes are stored in the serial register SSPSR. Each data is, after receiving the last eighth bit, loaded to the SSPBUF register from where it can be read. By reading this register, the acknowledge bit is automatically sent, which means that master device is ready to receive new data.

At the end, similar to data transmission, data reception ends by setting the STOP bit:

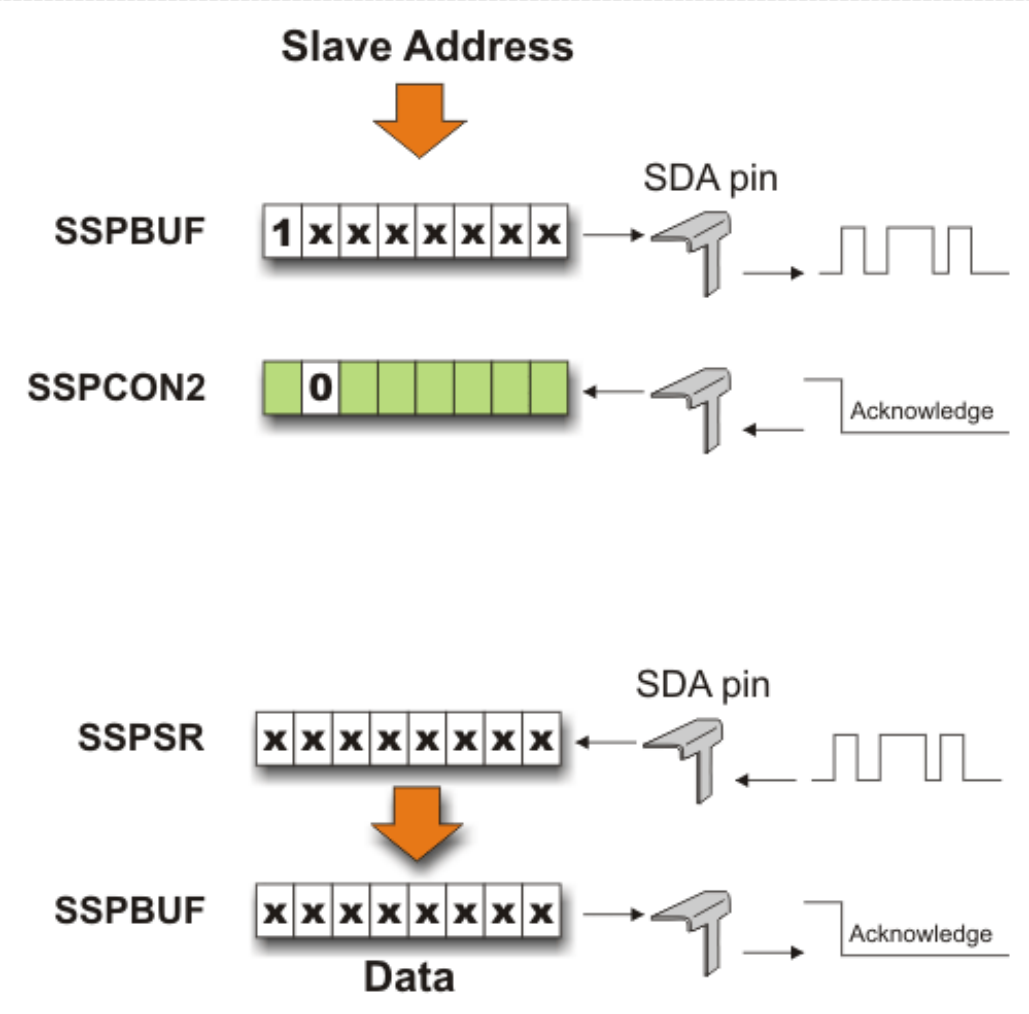


Fig. 6-30 Data Reception in I²C Master Mode

Start - Address - Acknowledge - Data - Acknowledge .... Data - Acknowledge - Stop!

In this pulse condition, the acknowledge bit is sent to slave device.

Baud Rate Generator

In order to synchronize data transmission, all events taking place on the SDA pin must be synchronized with the clock generated in master device. This clock is generated by a simple oscillator whose frequency depends on the microcontroller's main oscillator frequency, value written to the SSPADD register and the current SPI mode.

The clock frequency of the mode described in this book depends on selected quartz crystal and the SPADD register. The formula used to calculate it is shown in figure below.

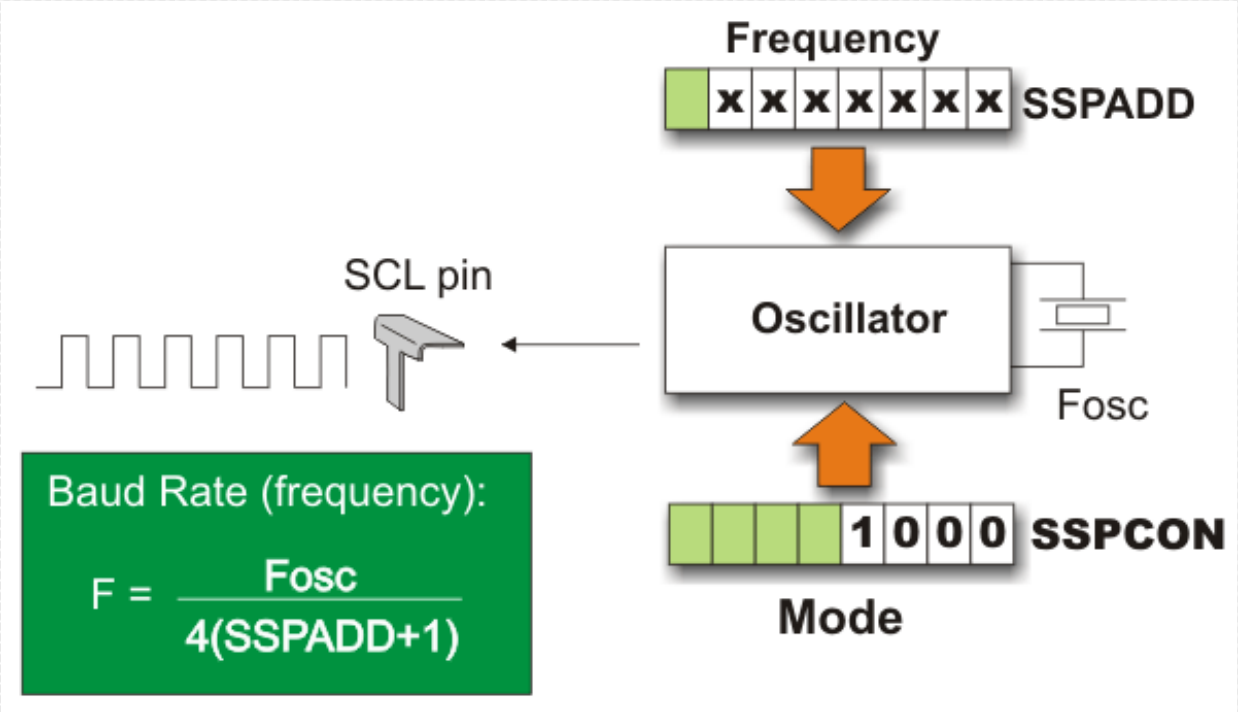


Fig. 6-31 Baud Rate Generator

Useful notes ...

When the microcontroller communicates with peripheral components, it may happen that data transfer fails for some reason. In that case, it is recommended to check the status of some bits which can clarify the problem. In practice, the state of these bits is checked by executing a short subroutine after each byte transmission and reception (just in case).

WCOL (SPCON,7) - If you try to write a new data to the SSPBUF register while another data transmit/receive is in progress, the WCOL bit will be set and the contents of the SSPBUF register remains unchanged. Write does not occur. After this, the WCOL bit must be cleared in software.

BF (SSPSTAT,0) - In transmit mode, this bit is set when the CPU writes to the SSPBUF register and remains set until the byte in serial format is shifted from the SSPSR register. In receive mode, this bit is set when data or address is loaded to the SSPBUF register. It is cleared when the SSPBUF register is read.

SSPOV (SSPCON,6) - In receive mode, this bit is set when a new byte is received by the SSPSR register via serial communication, whereas the previously received data has not been read from the SSPBUF register yet.

SDA and SCL Pins - When SPP module is enabled, these pins turns into Open Drain outputs. It means that these pins must be connected to the resistors which, at the other end, are connected to positive power supply.

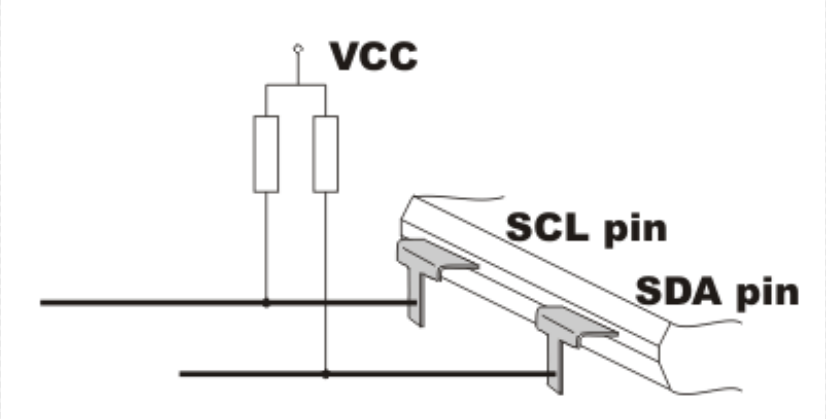


Fig. 6-32 Open Drain Output Resistors

In Short:

In order to establish serial communication in I²C mode, the following should be done:

Setting Module and Sending Address:

- Value to determine baud rate should be written to the SSPADD register;
- SlewRate control should be turned off by setting the SMP bit of the SSPSTAT register;
- In order to select Master mode, binary value 1000 should be written to the SSPM3-SSPM0 bits of the SSPCON1 register;
- The SEN bit of the SSPCON2 register should be set (START condition);
- The SSPIF bit is automatically set at the end of START condition when the module is ready to operate. It should be cleared;
- Slave address should be written to the SSPBUF register; and
- When the byte is sent, the SSPIF bit (interrupt) is automatically set when the acknowledge bit has been received from the Slave device.

Data Transmit:

- Data is to be send should be written to the SSPBUF register;
- When the byte is sent, the SSPIF bit (interrupt) is automatically set upon the acknowledge bit has been received from Slave device; and
- In order to inform the Slave device that transmit is complete, STOP condition should be initiated by setting the PEN bit of the SSPCON register.

Data Receive:

- In order to enable receive the RSEN bit of the SSPCON2 register should be set;
- The SSPIF bit signals data receive. When data is read from the SSPBUF register, the ACKEN bit of the SSPCON2 register should be set in order to enable sending acknowledge bit; and
- In order to inform Slave device that transmit is complete, the STOP condition should be initiated by setting the PEN bit of the SSPCON register.

Previous Chapter | Table of Contents | Next Chapter

Chapter 7: Analog Modules

Apart from a large number of digital I/O lines, the PIC16F887 contains 14 analog inputs. They enable the microcontroller to recognize, not only whether a pin is driven to logic zero or one (0 or +5V), but to precisely measure its voltage and convert it into a numerical value, i.e. digital format. The whole procedure takes place in the A/D converter module which has the following features:

- The converter generates a 10-bit binary result using the method of successive approximation and stores the conversion results into the ADC registers (ADRESL and ADRESH);
- There are 14 separate analog inputs;
- The A/D converter allows conversion of an analog input signal to a 10-bit binary representation of that signal; and
- By selecting voltage references Vref- and Vref+, the minimal resolution or quality of conversion may be adjusted to various needs.

ADC Mode and Registers

Even though the use of A/D converter seems to be very complicated, it is basically very simple, simpler than using timers and serial communication module, anyway.

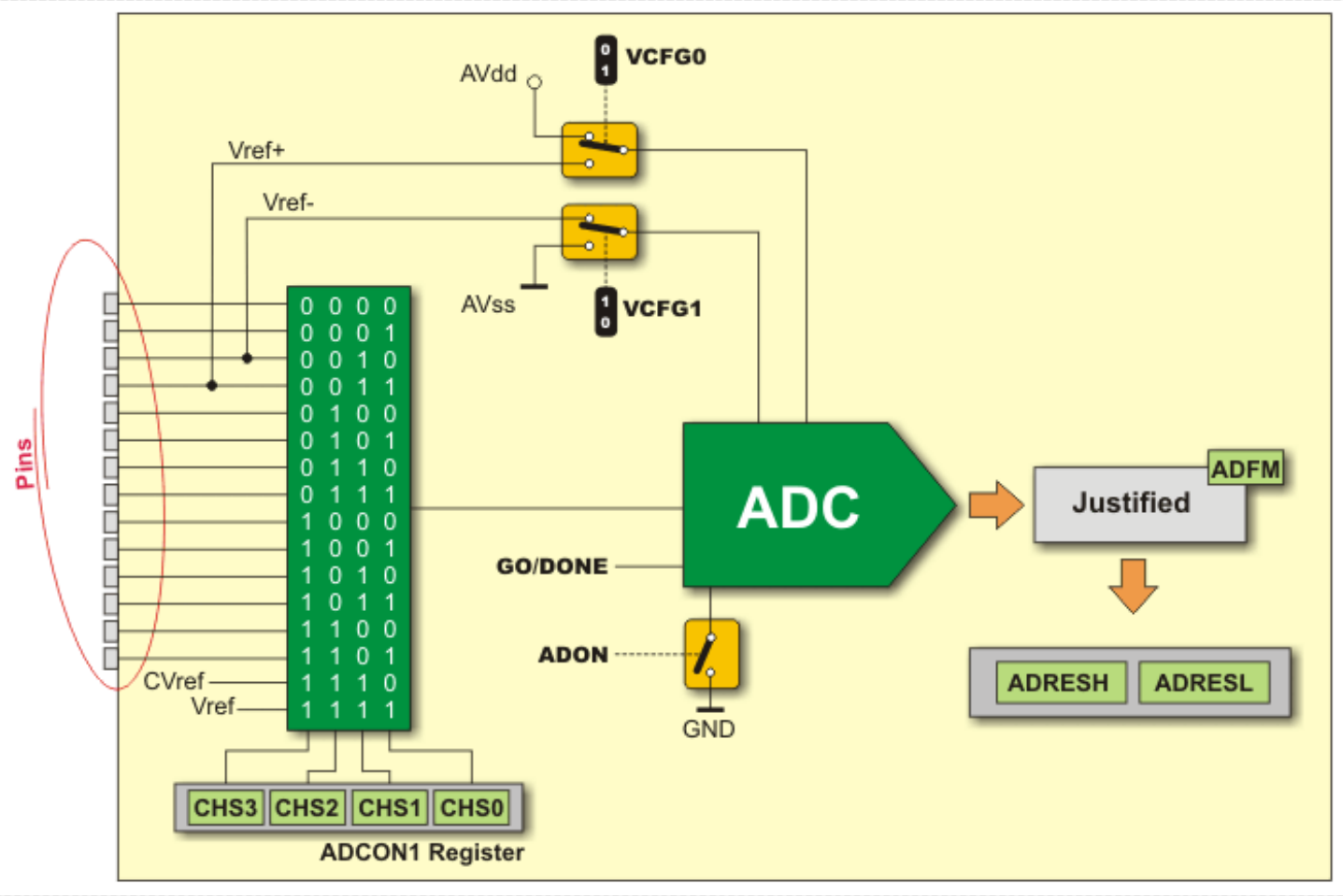


Fig. 7-1 ADC Mode and Registers

The module is under the control of the bits of four registers:

- ADRESH - Contains high byte of conversion result;
- ADRESL - Contains low byte of conversion result;
- ADCON0 - Control register 0; and
- ADCON1 Control register 1

ADRESH and ADRESL Registers

When converting an analog value into a digital one, the result of the 10-bit A/D conversion will be stored in these two registers. In order to deal with this value easier, it can appear in two formats- left justified and right justified. The ADFM bit of the ADCON1 register determines the format of conversion result (see figure 7-2). In the event that A/D converter is not used, these registers may be used as general-purpose registers.

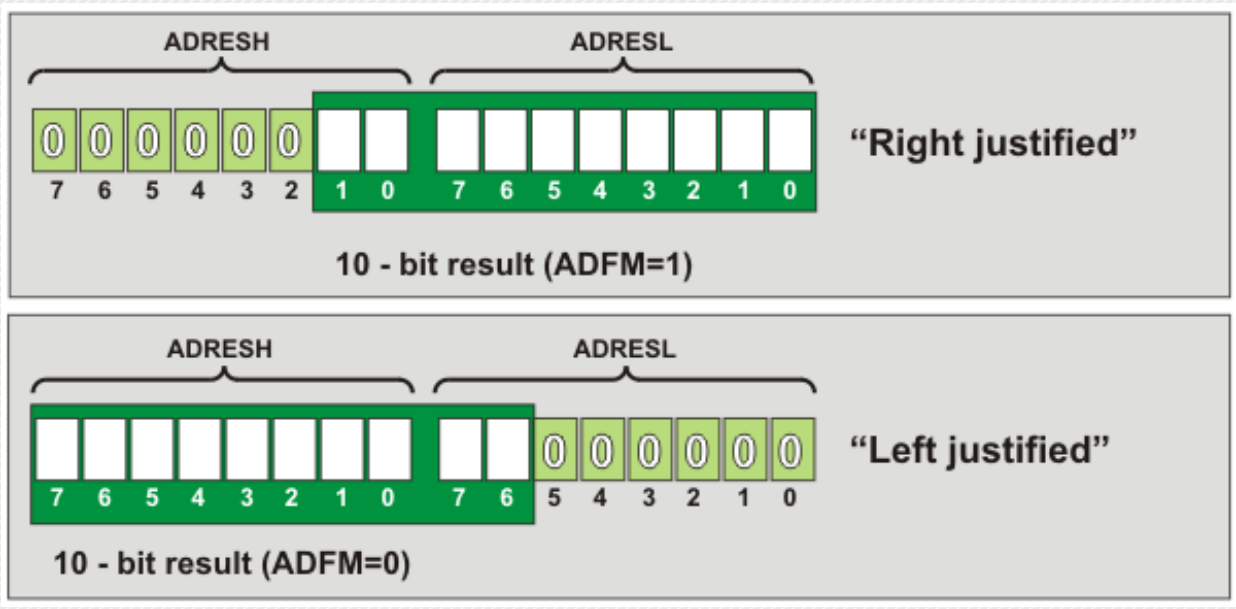


Fig. 7-2 ADRESH and ADRESL Registers

A/D Acquisition Requirements

For the ADC to meet its specified accuracy, it is necessary to provide a certain time delay between selecting specific analog input and measurement itself. This time is called "acquisition time" and mainly depends on the source impedance. There is an equation used for accurately calculating this time, which in the worst case amounts to approximately 20uS. Briefly, after selecting (or changing) the analog input and before starting conversion it is necessary to provide at least 20uS time delay to enable the ACD maximal conversion accuracy.

ADC Clock Period

Time needed to complete a one-bit conversion is defined as TAD. The required TAD must be at least 1,6 uS. One full 10-bit A/D conversion is a bit longer than expected and amounts to 11 TAD periods. However, since both the conversion clock frequency and source are determined by software, one of the available combination of bits ADCS1 and ADCS0 should be selected before voltage measurement on some analog input starts. These bits are stored in the ADCON0 register.

ADC Clock Source	ADCS1	ADCS0	Device Frequency (Fosc)			
			20 Mhz	8 Mhz	4 Mhz	1 Mhz
Fosc/2	0	0	100 nS	250 nS	500 nS	2 uS
Fosc/8	0	1	400 nS	1 uS	2 uS	8 uS
Fosc/32	1	0	1.6 uS	4 uS	8 uS	32 uS
Frc	1	1	2 - 6 uS	2 - 6 uS	2 - 6 uS	2 - 6 uS

Table 7-1 ADC Clock Period

Any change in the system clock frequency will affect the ADC clock frequency, which may adversely affect the ADC result. Device frequency characteristics are shown in the table above. The values in the shaded cells are outside of recommended range.

How to Use A/D Converter?

In order to enable the A/D converter to run without problems as well as to avoid unexpected results, it is necessary to consider the following:

- A/D converter does not differ between digital and analog voltages. In order to avoid errors in measurement or chip damage, the pins should be configured as analog inputs before conversion starts. The bits used for this purpose are stored in the TRIS and ANSELH registers;
- When the port with analog inputs marked as CH0-CH13 is read, the corresponding bits will be driven to logic zero (0); and

- Roughly speaking, voltage measurement in the converter is based on comparing input voltage with internal scale which has 1024 marks (210=1024). The lowest scale mark stands for the Vref- voltage, whilst the highest mark stands for the Vref+ voltage. Figure 7-3 below shows selectable referent voltages and their minimum and maximum values as well.

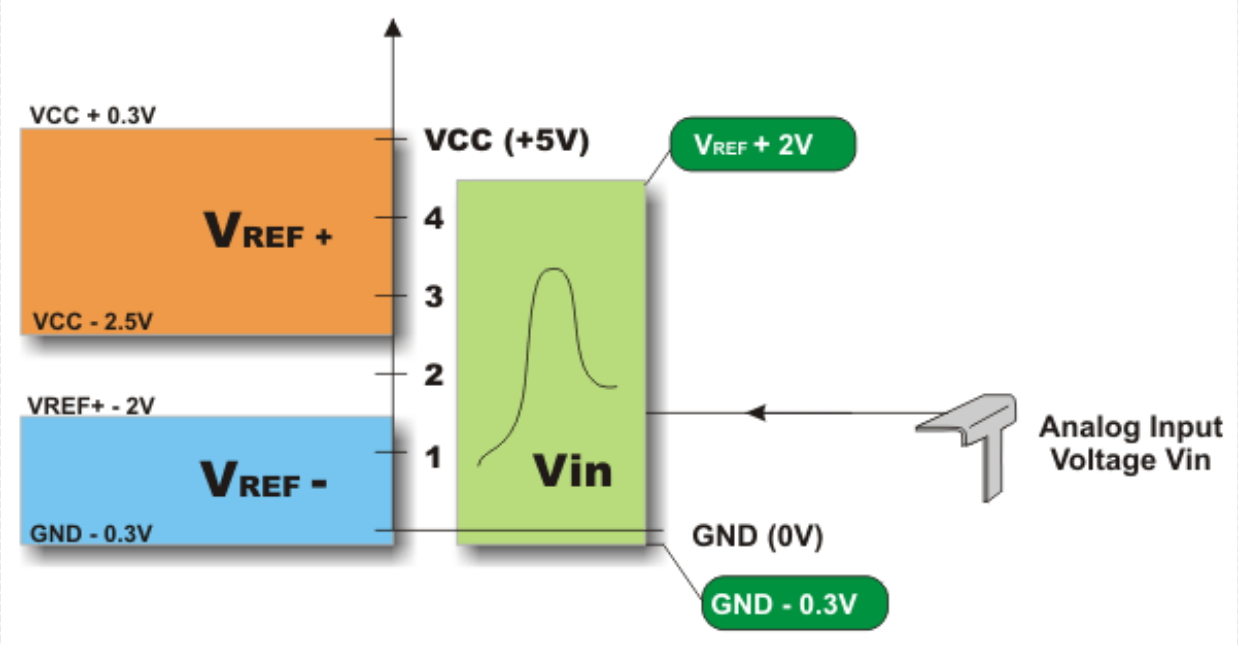


Fig. 7-3 How to Use The A/D Converter

ADCON0 Register

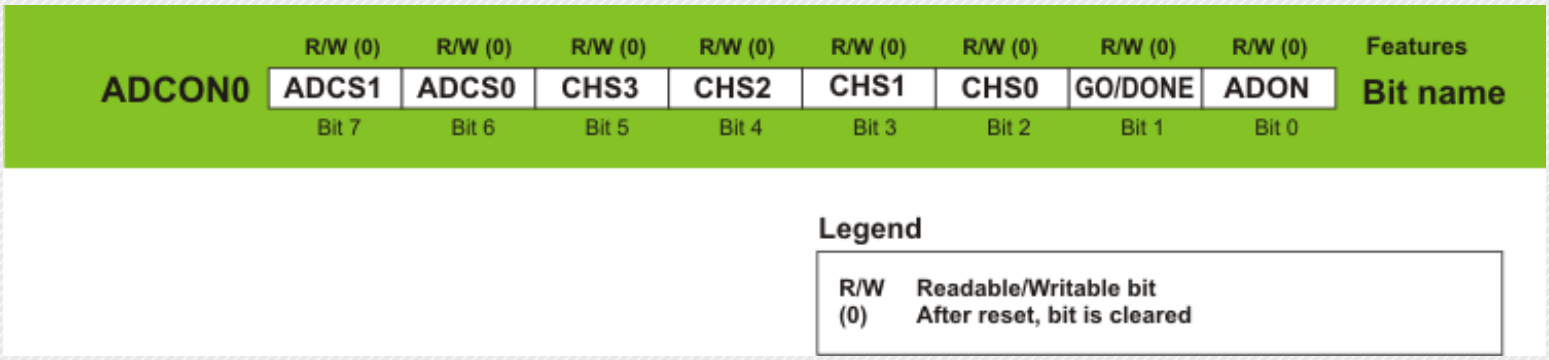


Fig. 7-4 ADCON0 Register

ADCS1, ADCS0 - A/D Conversion Clock Select bits select clock frequency used for internal synchronization of A/D converter. It also affects duration of conversion.

ADCS1	ADCS2	Clock
0	0	Fosc/2
0	1	Fosc/8
1	0	Fosc/32
1	1	RC *

Table 7-2 A/D Conversion Select Bits

* Clock is generated by internal oscillator which is built in converter.

CHS3-CHS0 - Analog Channel Select bits select a pin or an analog channel for conversion, i.e. voltage measurement:

CHS3	CHS2	CHS1	CHS0	Channel	Pin
0	0	0	0	0	RA0/AN0
0	0	0	1	1	RA1/AN1
0	0	1	0	2	RA2/AN2
0	0	1	1	3	RA3/AN3
0	1	0	0	4	RA5/AN4
0	1	0	1	5	RE0/AN5
0	1	1	0	6	RE1/AN6
0	1	1	1	7	RE2/AN7
1	0	0	0	8	RB2/AN8
1	0	0	1	9	RB3/AN9
1	0	1	0	10	RB1/AN10
1	0	1	1	11	RB4/AN11
1	1	0	0	12	RB0/AN12
1	1	0	1	13	RB5/AN13
1	1	1	0	CVref	
1	1	1	1	Vref = 0.6V	

Table 7-3 Analog Channel Status Bits

GO/DONE - A/D Conversion Status bit determines current status of conversion:

- 1 - A/D conversion is in progress; and
- 0 - A/D conversion is complete. This bit is automatically cleared by hardware when the A/D conversion is completed.

ADON - A/D On bit enables A/D converter.

- 1 - A/D converter is enabled; and
- 0 - A/D converter is disabled.

ADCON1 Register

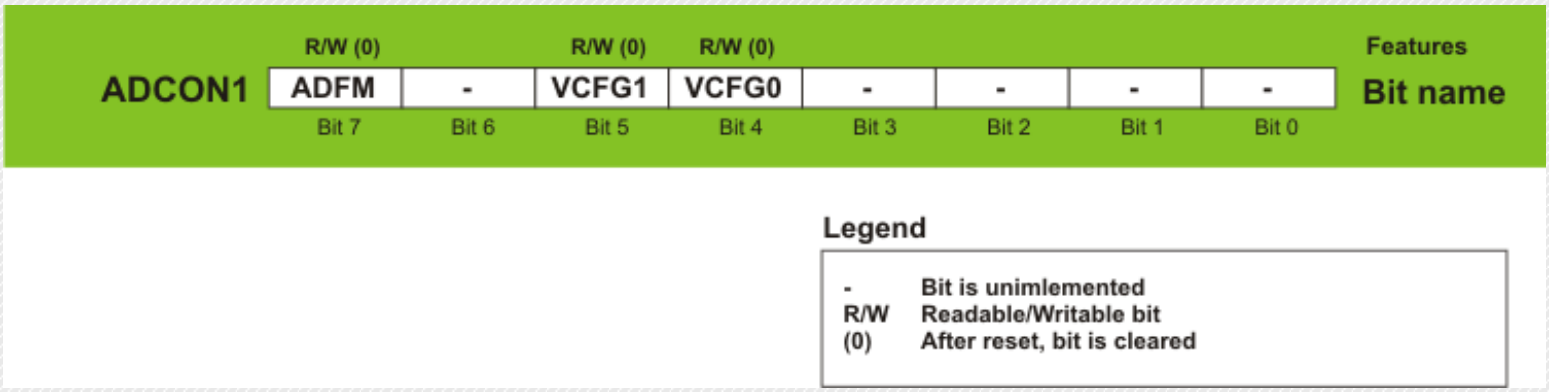


Fig. 7-5 ADCON1 Register

ADFM - A/D Result Format Select bit

- 1 - Conversion result right justified. Six most significant bits of the ADRESLH are not used; and
- 0 - Conversion result left justified. Six least significant bits of the ADRESL are not used.

VCFG1 - Voltage Reference bit selects negative voltage reference source needed for A/D converter operating.

- 1 - Negative voltage reference is applied on the Vref- pin; and
- 0 - Voltage power supply Vss is used as negative voltage reference source.

VCFG0 - Voltage Reference bit selects positive voltage reference source needed for A/D converter operating.

- 1 - Positive voltage reference is applied on the Vref+ pin; and
- 0 - Voltage power supply Vdd is used as positive voltage reference source.

In Short:

In order to measure voltage on an input pin by A/D converter the following should be done:

Step 1 - Configuring port:

- Write logic one (1) to the corresponding bit of the TRIS register to configure it as input; and
- Write logic one (1) to the corresponding bit of the ANSEL register to configure it as analog input.

Step 2 - Configuring ADC module:

- Configure voltage reference in the ADCON1 register;
- Select ADC conversion clock in the ADCON0 register;
- Select one of input channels CH0-CH13 of the ADCON0 register;
- Select data format using the ADFM bit of the ADCON1 register; and
- Enable A/D converter by setting the ADON bit of the ADCON0 register.

Step 3 - Configuring ADC interrupt (optionally):

- Clear the ADIF bit; and
- Set the ADIE, PEIE and GIE bits.

Step 4 - Wait for the required acquisition time (approximately 20uS) to pass.

Step 5 - Start conversion by setting the GO/DONE bit of the ADCON0 register.

Step 6 - Wait for ADC conversion to complete.

- It is necessary to check in program loop whether the GO/DONE pin is cleared or wait for an A/D interrupt (must be previously enabled).

Step 7 - Read ADC results:

- Read the ADRESH and ADRESL registers.

Analog Comparator

In addition to A/D converter, there is one more module, which until quite recently has been embedded only in integrated circuits, belonging to so called analog electronics. Owing to the fact that it is hardly possible to find any more complex automatic device which in some way does not use these circuits, two high quality comparators along with additional electronics are integrated into the microcontroller and connected to its pins.

How does a comparator operate? Basically, the analog comparator is an amplifier which compares the magnitude of voltages at two inputs. Looking at its physical features, it has two inputs and one output. Depending on which input has a higher voltage (analog value), a logic zero (0) or logic one (1) (digital values) will appear on its output:

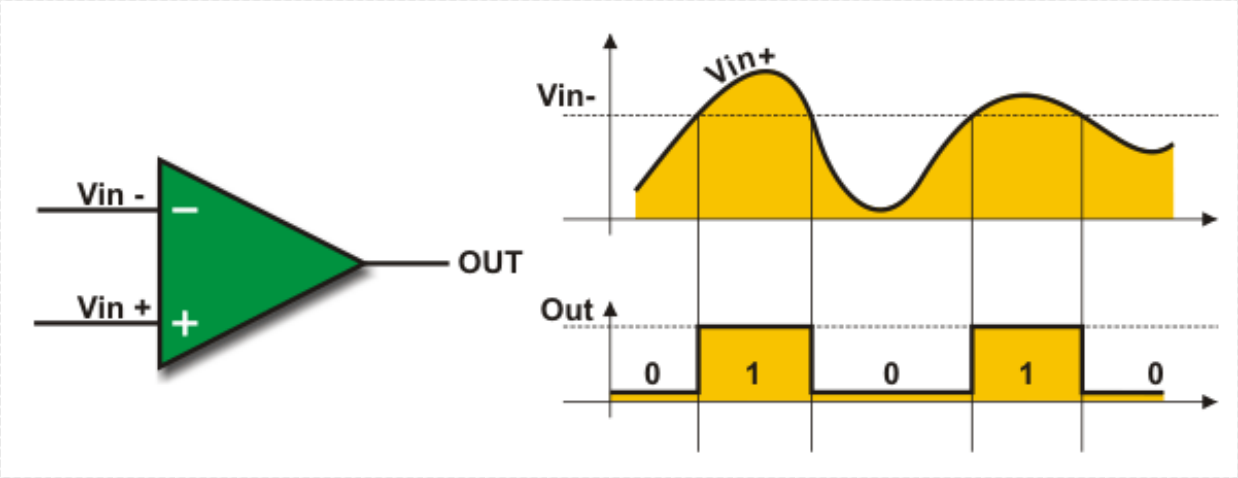


Fig. 7-6 Analog Comparator

- When the analog voltage at Vin- is higher than the analog voltage at Vin+, the output of the comparator is a digital low level; and
- When the analog voltage at Vin+ is higher than the analog voltage at Vin-, the output of the comparator is a digital high level.

The PIC16F887 microcontroller has two such voltage comparators whose inputs are connected to I/O pins RA0-RA3, whereas the outputs are connected to the pins RA4 and RA5. In addition there is also a referent voltage internal source on chip itself, but it will be discussed later.

These two circuits are under control of the bits stored in the following registers:

CM1CON0 is in control of comparator C1;
CM2CON0 is in control of comparator C2; and
CM2CON1 is in control of comparator C2.

Voltage Reference Internal Source

One of two analog voltages provided on the comparator inputs is usually stable and unchangeable. Because of those features it is called "voltage reference"(Vref). To generate it, both external and special internal voltage source can be used. After selecting voltage source, Vref is derived from it by means of ladder network consisting of 16 resistors which form voltage divider. The voltage source is selectable through both ends of that divider through the VRSS bit of the VRCON register.

In addition, the voltage fraction provided by resistor ladder network may be selected through the bits VR0-VR3 and used as voltage reference. See figure below.

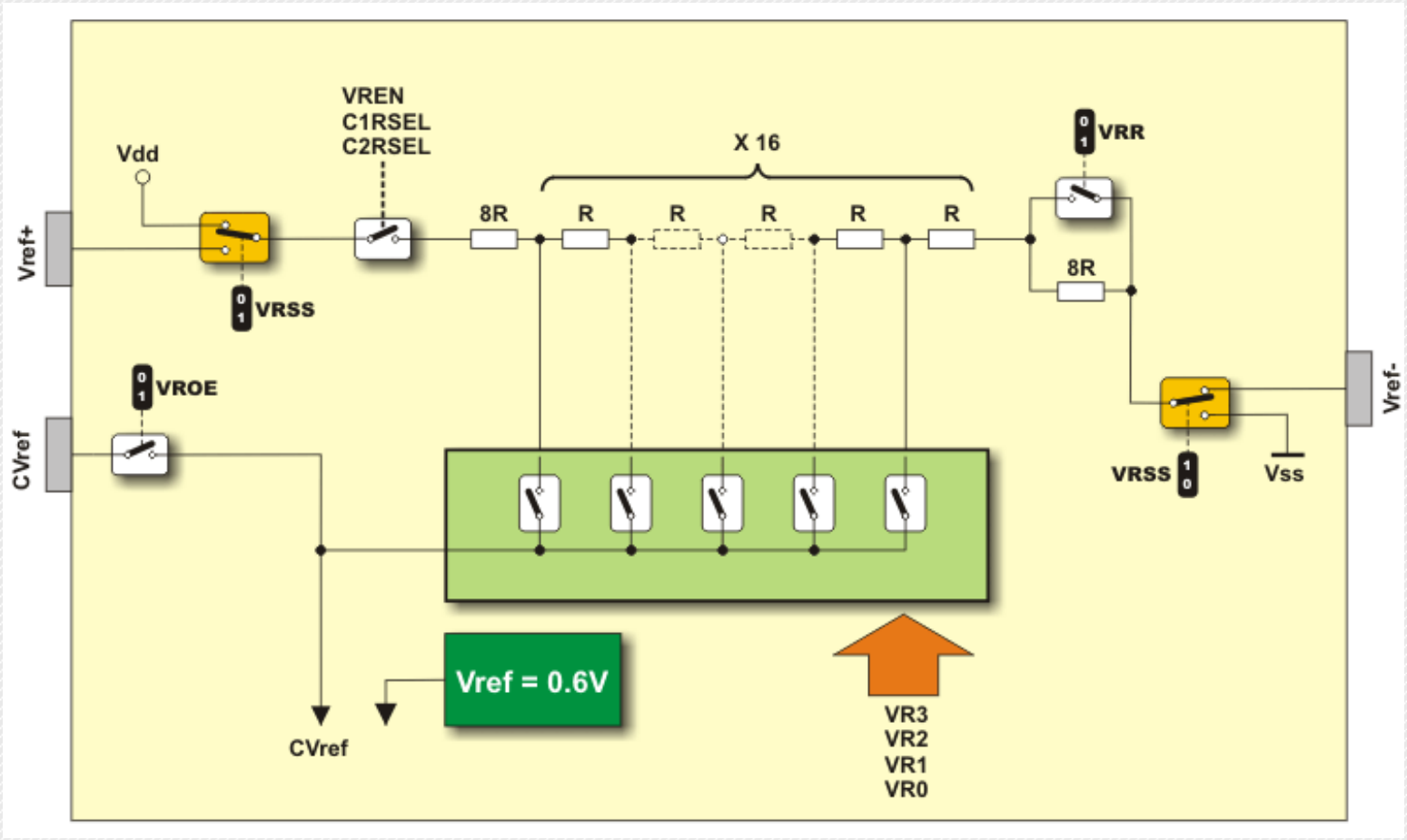


Fig. 7-7 VREF

The comparator voltage reference has 2 ranges with 16 voltage levels in each range. Range selection is controlled by the VRR bit of the VRCON register. The selected voltage reference may be output to the RA2/AN2 pin.

Even though the main idea was to obtain varying voltage reference for the operation of analog modules, a simple A/D converter is obtained in that way too. This converter is very useful in some situations.

It's operation is under control of the VRCON register.

Comparators and Interrupt Operation

The flag bit CMIF of the register PIR is set on every change of logic state on any comparator's output. The same changes also cause an interrupt if the following bits are set:

CMIE bit of the PIE register;
PEIE bit of the INTCON register; and
GIE bit of the INTCON register.

If interrupt is enabled, any change on the comparator's output can wake up the microcontroller from *sleep mode* if it is setup in that mode.

CM1CON0 Register

	R/W (0)	R (0)	R/W (0)	R/W (0)		R/W (0)	R/W (0)	R/W (0)	Features
CM1CON0	C1ON	C1OUT	C1OE	C1POL	-	C1R	C1CH1	C1CH0	Bit name
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	

Legend

-	Bit is unimplemented
R/W	Readable/Writable bit
R	Readable bit
(0)	After reset, bit is cleared

Fig. 7-8 CM1CON0 Regsiter

Bits of this register are in control of the comparator C1. It mainly affects configuration of its inputs. To understand it better, look at figure 7-9 below which shows only a part of electronics directly affected by the bits of this register.

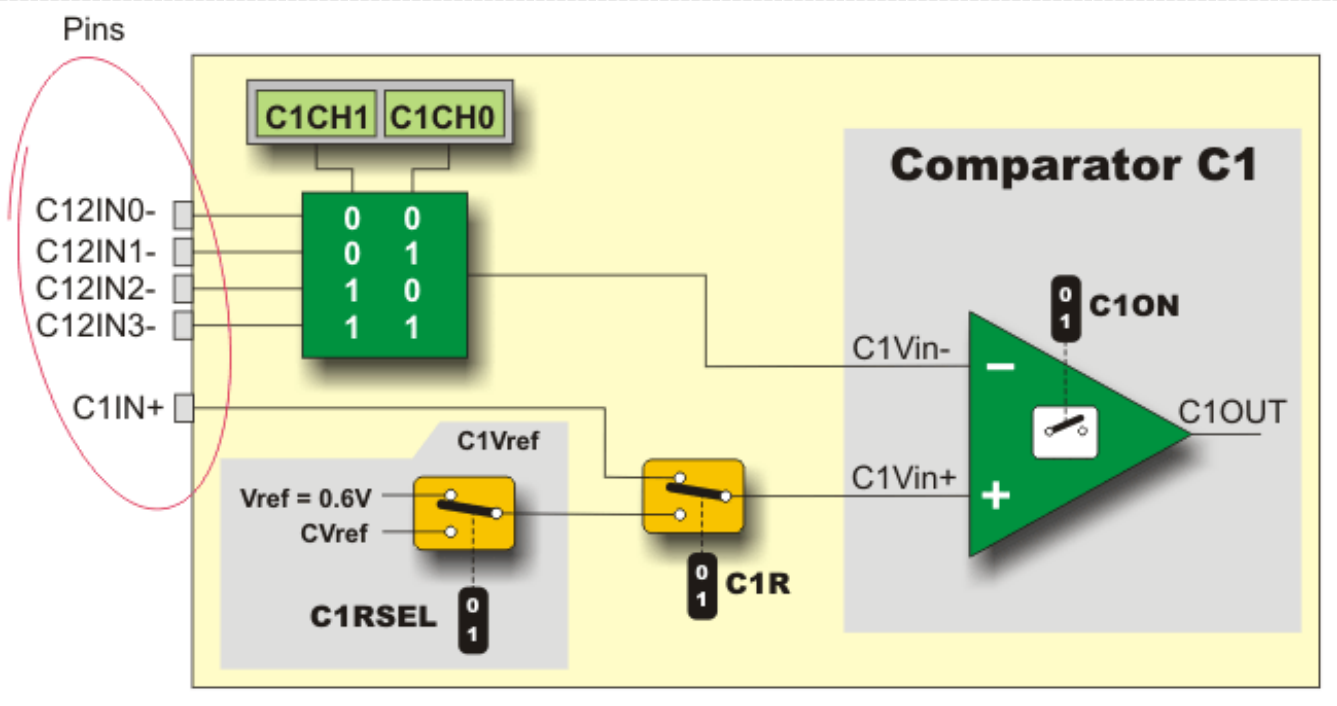


Fig. 7-9 Comparator C1 Enable Bit

C1ON - Comparator C1 Enable bit enables comparator C1.

- 1 - Comparator C1 is enabled; and
- 0 - Comparator C1 is disabled.

C1OUT - Comparator C1 Output bit is comparator C1 output bit.

If C1POL = 1 (comparator output is inverted)
- 1 - Analog voltage at C1Vin+ is lower than analog voltage at C1Vin-; and
- 0 - Analog voltage at C1Vin+ is higher than analog voltage at C1Vin-.

If C1POL = 0 (comparator output is non-inverted)
- 1 - Analog voltage at C1Vin+ is higher than analog voltage at C1Vin-; and
- 0 - Analog voltage at C1Vin+ is lower than analog voltage at C1Vin-.

C1OE Comparator C1 Output Enable bit.

- 1 - Comparator C1OUT output is connected to the C1OUT pin.*; and
- 0 - Comparator output is internal only.

* In order to enable the C1OUT bit to be present on the pin, two conditions must be met: C1ON = 1 (comparator must be on) and the corresponding TRIS bit = 0 (pin must be configured as output).

C1POL - Comparator C1 Output Polarity Select bit enables comparator C1 out put state to be inverted.

- 1 - Comparator C1 output is inverted; and
- 0 - Comparator C1 output is non-inverted.

C1R - Comparator C1 Reference Select bit
- 1 - Non-inverting input C1Vin+ is connected to reference voltage C1Vref; and
- 0 - Non-inverting input C1Vin+ is connected to the C1IN+ pin.

C1CH1, C1CH0 - Comparator C1 Channel Select bit

C1CH1	C1CH0	Comparator C1Vin- input
0	0	Input C1Vin- is connected to the C12IN0- pin
0	1	Input C1Vin- is connected to the C12IN1- pin
1	0	Input C1Vin- is connected to the C12IN2- pin
1	1	Input C1Vin- is connected to the C12IN3- pin

Table 7-4 Comparator C1

CM2CON0 Register

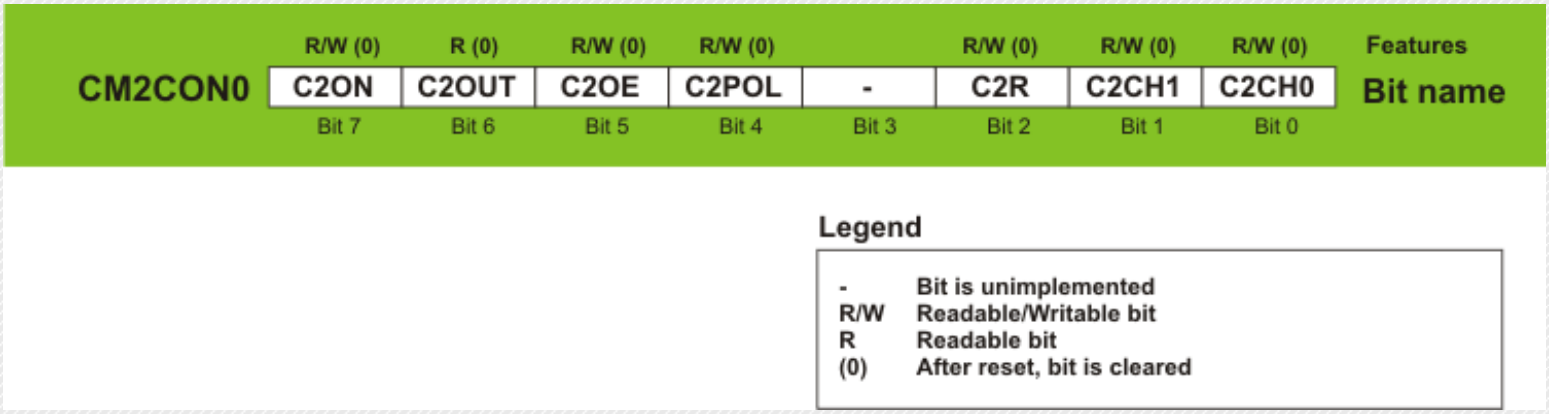


Fig. 7-10 CM2CON0 Regsiter

Bits of this register are in control of comparator C2. Similar to the previous case, the figure 7-11 shows a simplified schematic of the circuit affected by the bits of this register.

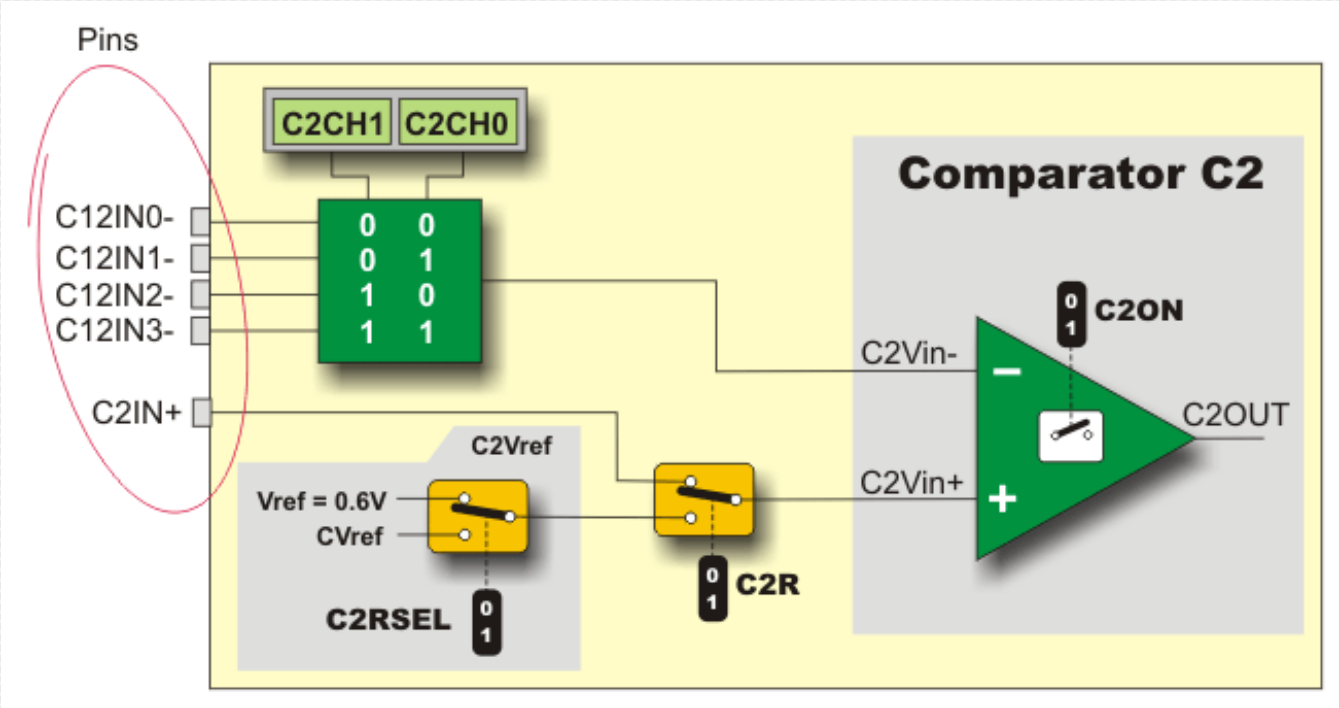


Fig. 7-11 Comparator C2 Schematic Diagram

C2ON - Comparator C2 Enable bit enables comparator C2.

- 1 - Comparator C2 is enabled; and
- 0 - Comparator C2 is disabled.

C2OUT - Comparator C2 Output bit is comparator C2 output.

If C2POL = 1 (comparator output inverted)
- 1 - Analog voltage at C1Vin+ is lower than analog voltage at C1Vin-; and
- 0 - Analog voltage at C1Vin+ is higher than analog voltage at C1Vin-.

If C2POL = 0 (comparator output non-inverted)
- 1 - Analog voltage at C1Vin+ is higher than analog voltage at C1Vin-; and
- 0 - Analog voltage at C1Vin+ is lower than analog voltage at C1Vin-.

C2OE - Comparator C2Output Enable bit

- 1 - Comparator C2OUT output is connected to the C2OUT pin.*; and
- 0 - Comparator output is internal only.

* In order to enable the C2OUT bit to be present on the pin, two conditions must be met: C2ON = 1 (comparator must be on) and the corresponding TRIS bit = 0 (pin must be configured as output).

C2POL - Comparator C2 Output Polarity Select bit enables comparator C2 out put state to be inverted.

- 1 - Comparator C2 output is inverted; and
- 0 - Comparator C2 output is non-inverted.

C2R - Comparator C2 Reference Select bit

- 1 - Non-inverting input C2Vin+ is connected to reference voltage C2Vref; and
- 0 - Non-inverting input C2Vin+ is connected to the C2IN+ pin.

C2CH1, C2CH0 Comparator C2 Channel Select bit

C2CH1	C2CH0	Comparator C2Vin- input
0	0	Input C2Vin- is connected to the C12IN0- pin
0	1	Input C2Vin- is connected to the C12IN1- pin
1	0	Input C2Vin- is connected to the C12IN2- pin
1	1	Input C2Vin- is connected to the C12IN3- pin

Table 7-5 Comparator C2 Channel Select Bit

CM2CON1 Register

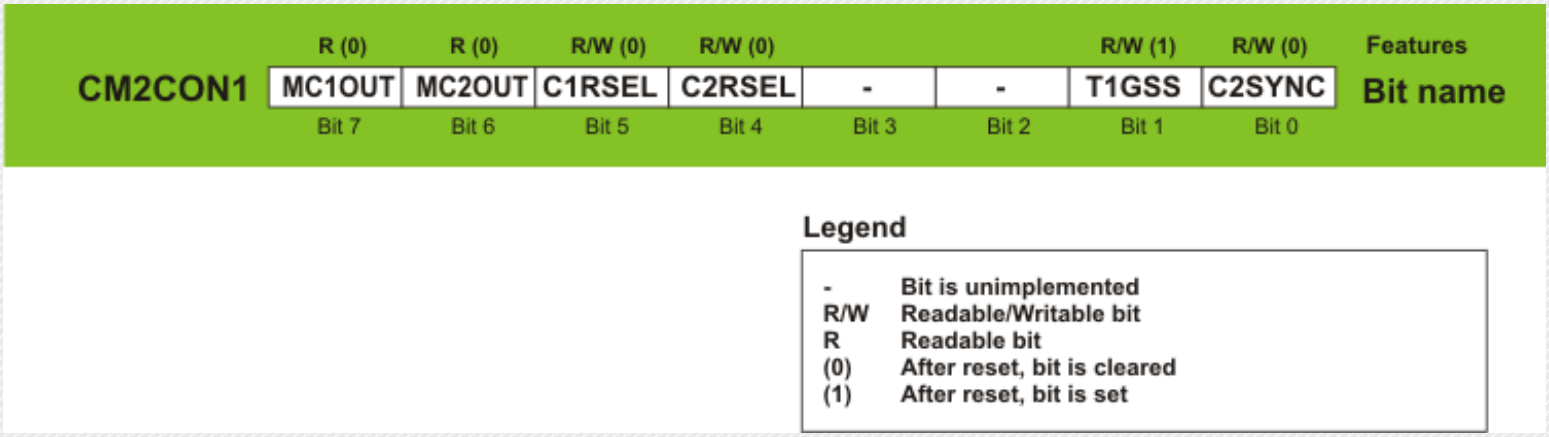


Fig. 7-12 CM2CON1 Register

MC1OUT Mirror Copy of C1OUT bit

MC2OUT Mirror Copy of C2OUT bit

C1RSEL Comparator C1 Reference Select bit

- 1 - Selectable voltage CVref is used in voltage reference C1Vref source; and
- 0 - Fixed voltage reference 0.6V is used in voltage reference C1Vref source.

C2RSEL - Comparator C2 Reference Select bit

- 1 - Selectable voltage CVref is used in voltage reference C2Vref source; and
- 0 - Fixed voltage reference 0.6V is used in voltage reference C2Vref source.

T1GSS - Timer1 Gate Source Select bit

- 1 - Timer T1gate source is T1G; and
- 0 - Timer T1gate source is comparator SYNCC2OUT.

C2SYNC - Comparator C2 Output Synchronization bit

- 1 - Comparator C2 output is synchronized to falling edge of Timer TMR1 clock; and
- 0 - Comparator output is asynchronous signal.

VRCON Register

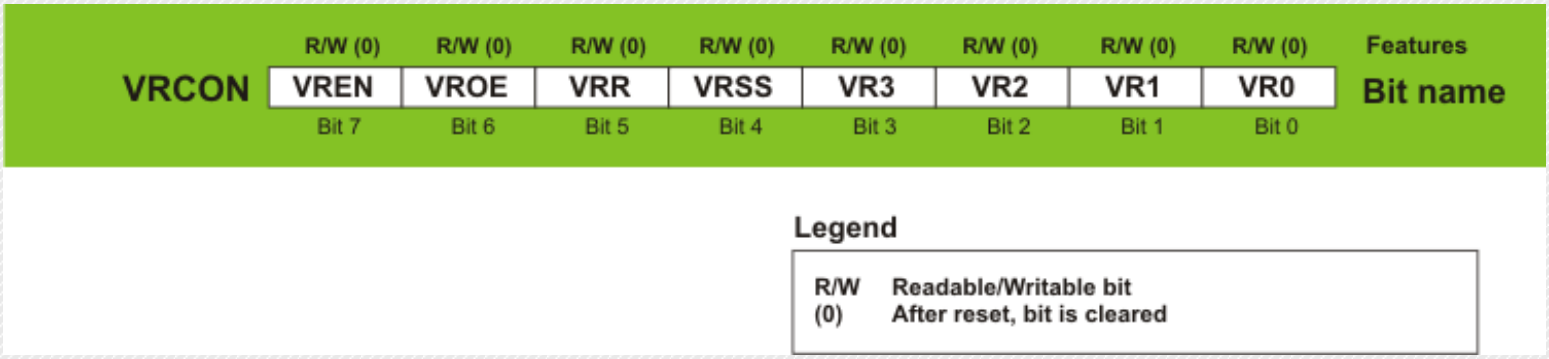


Fig. 7-13 VRCON Register

VREN Comparator C1 Voltage Reference Enable bit

- 1 - Voltage reference CVref source is powered on; and
- 0 - Voltage reference CVref source is powered off.

VROE Comparator C2 Voltage Reference Enable bit

- 1 - Voltage reference CVref is connected to the pin; and
- 0 - Voltage reference CVref is disconnected from the pin.

VRR - CVref Range Selection bit

- 1 - Voltage reference source is set to low range; and
- 0 - Voltage reference source is set to high range.

VRSS - Comparator Vref Range selection bit

- 1 - Voltage reference source is in the range of Vref+ to Vref-; and
- 0 - Voltage reference source is in the range of Vdd - Vss (power supply voltage).

VR3 - VR0 CVref Value Selection

If VRR = 1 (low range)

Voltage reference is calculated using the formula: CVref = ([VR3:VR0]/24)Vdd

If VRR = 0 (high range)

Voltage reference is calculated using the formula: CVref = Vdd/4 + ([VR3:VR0]/32)Vdd

In Short:

In order to properly use built in Comparators, it is necessary to do the following:

Step 1 - Configuring module:

- In order to select the appropriate mode, bits of the registers CM1CON0 and CM2CON0 should be configured. Interrupt should be disabled on any change of mode.

Step 2 - Configuring internal voltage reference Vref source (only when used). In the VRCON register it is necessary to :

- Select one of two voltage ranges using the VRR bit;
- Configure necessary Vref using bits VR3 - VR0;
- Set the VROE bit if needed; and
- Enable voltage Vref source by setting the VREN bit.

Formula used to calculate voltage reference:

VRR = 1 (low range)
CVref = ([VR3:VR0]/24)VLADDER

VRR = 0 (high range)
CVref = (VLADDER/4) + ([VR3:VR0]VLADDER/32)

Vladder = Vdd or ([Vref+] - [Vref-]) or Vref+

Step 3 - Starting operation:

- Enable interrupt by setting bits CMIE (PIE register), PEIE and GIE (both in the INTCON register);
- Read bits C1OUT and C2OUT of the CMCON register; and
- Read flag bit CMIF of the PIR register. After being set, this bit must be cleared in software.

Previous Chapter | Table of Contents | Next Chapter

Chapter 8: Other MCU's Circuits

As seen in figure below, clock signal may be generated by one of two built in oscillators.

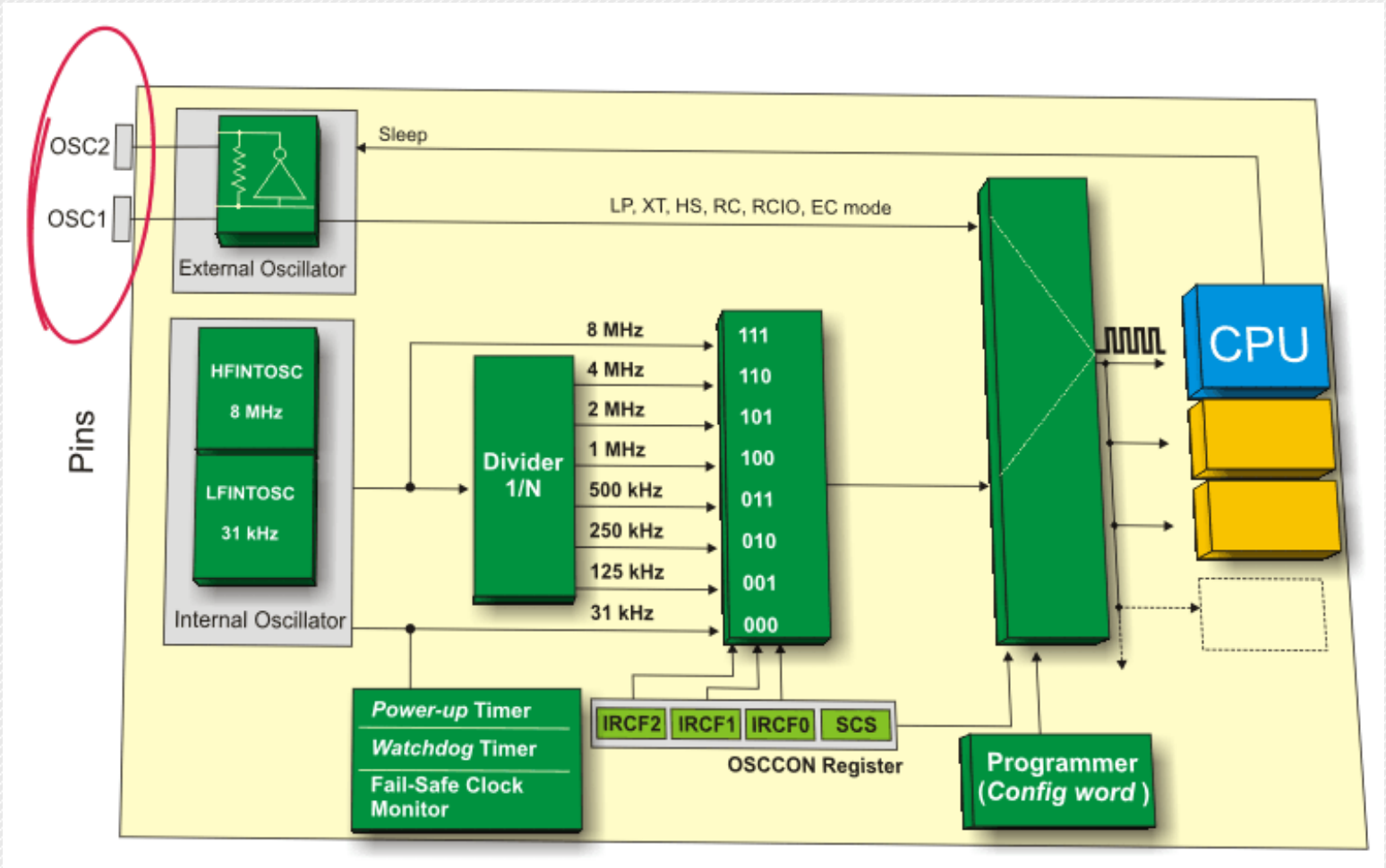


Fig. 8-1 Two built in oscillators

An **External oscillator** is installed within the microcontroller and connected to the OSC1 and OSC2 pins. It is called "external" because it relies on external circuitry for the clock signal and frequency stabilization, such as a stand-alone oscillator, quarts crystal, ceramic resonator or resistor-capacitor circuit. The oscillator mode is selected by bits of bytes sent during programming, so called **Config Word**.

Internal oscillator consists of two separate, internal oscillators:

The HFINTOSC is a high-frequency internal oscillator which operates at 8MHz. The microcontroller can use clock source generated at that frequency or after being divided in prescaler; and

The LFINTOSC is a low-frequency internal oscillator which operates at 31 kHz. Its clock sources are used for watch-dog and power-up timing but it can also be used as a clock source for the operation of the entire microcontroller.

The system clock can be selected between external or internal clock sources via the System Clock Select (SCS) bit of the OSCCON register.

OSCCON Register

The OSCCON register controls the system clock and frequency selection options. It contains the following bits: frequency selection bits (IRCF2, IRCF1, IRCF0), frequency status bits (HTS, LTS), system clock control bits (OSTA, SCS).

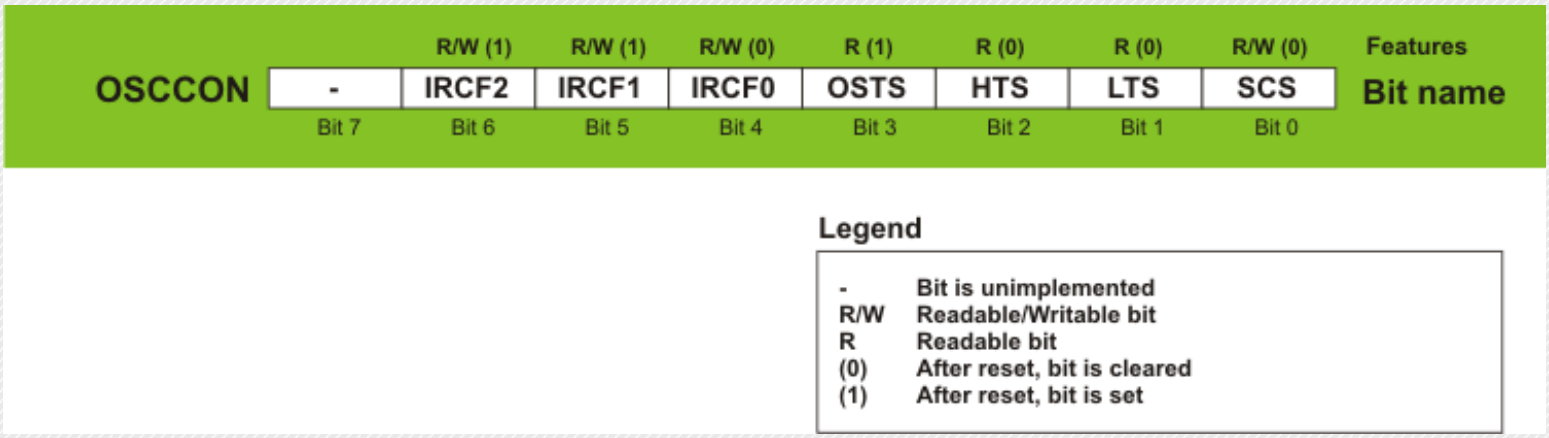


Fig. 8-2 OSCCON Register

IRCF2-0 - Internal Oscillator Frequency Select bits. Combination of these three bits determines the divider rate. The clock frequency of internal oscillator is also determined in this way.

IRCF2	IRCF1	IRCF0	Frequency	OSC.
1	1	1	8 MHz	HFINTOSC
1	1	0	4 MHz	HFINTOSC
1	0	1	2 MHz	HFINTOSC
1	0	0	1 MHz	HFINTOSC
0	1	1	500 kHz	HFINTOSC
0	1	0	250 kHz	HFINTOSC
0	0	1	125 kHz	HFINTOSC
0	0	0	31 kHz	LFINTOSC

Table 8-1 Internal Oscillator Frequency Select Bits

OSTS - Oscillator Start-up Time-out Status bit indicates which clock source is currently in use. This bit is readable only.

- 1 - External clock oscillator is in use; and
- 0 - One of internal clock oscillators is in use (HFINTOSC or LFINTOSC).

HTS - HFINTOSC Status bit (8 MHz - 125 kHz) indicates whether high-frequency internal oscillator operates in a stable way.

- 1 - HFINTOSC is stable; and
- 0 - HFINTOSC is not stable.

LTS - LFINTOSC Stable bit (31 kHz) indicates whether low-frequency internal oscillator operates in a stable way.

- 1 - LFINTOSC is stable; and
- 0 - LFINTOSC is not stable.

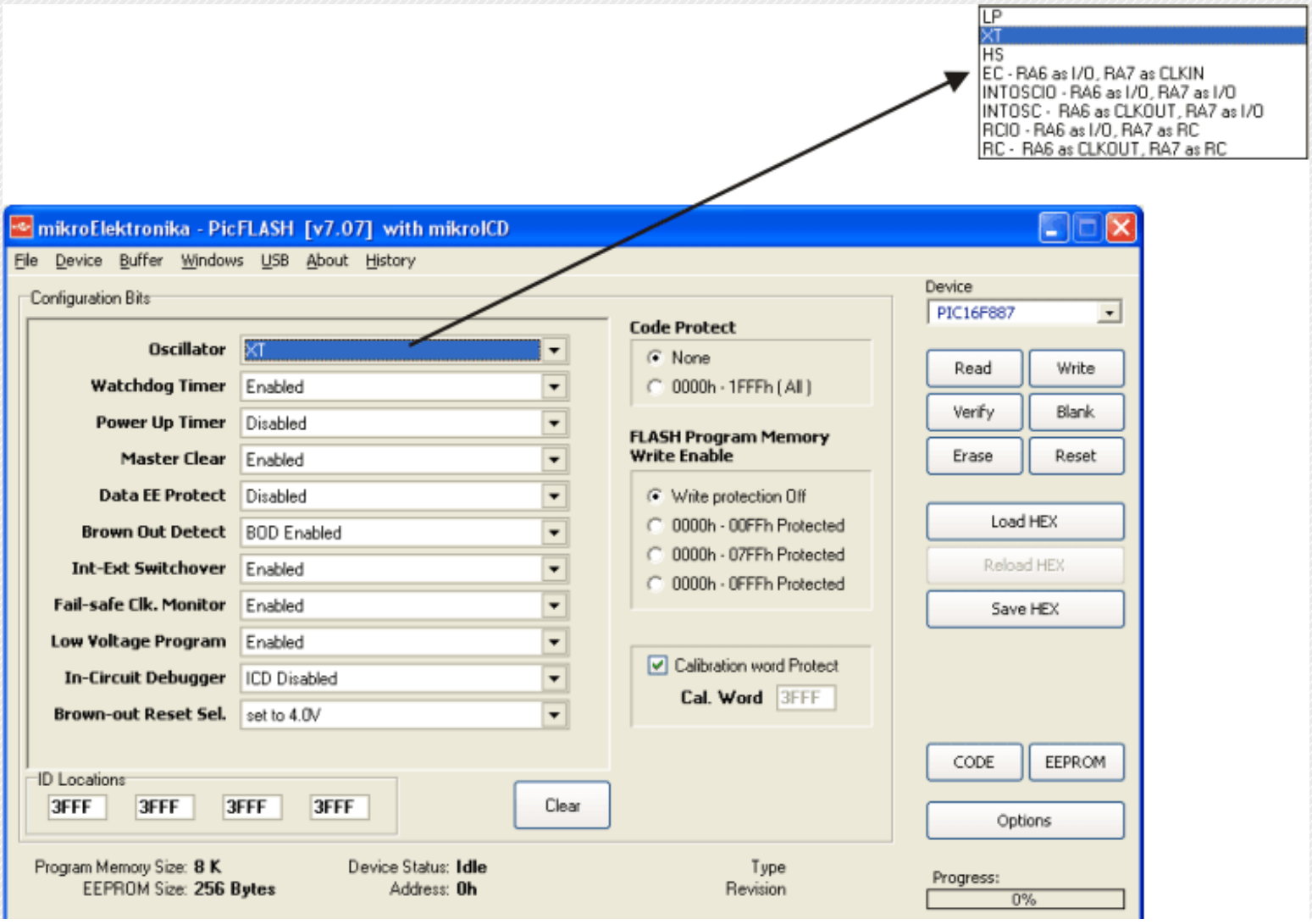
SCS - System Clock Select bit determines which oscillator is to be used as a clock source.

- 1 - Internal oscillator is used for system clock;
- 0 - External oscillator is used for system clock; and
  The oscillator mode is set by bits in Config Word which are written to the microcontroller memory during programming.

External Clock Modes

In order to enable the external oscillator to operate at different speeds and use different components for frequency stabilization, it can be configured to operate in one of several modes. Mode selection is performed after the program writing and compiling. First of all, it is necessary to activate the program on PC used for programming. In this case, PICflash. Click on the oscillator combox and select one option from the drop-down list. After that, the appropriate bits will be set becoming in that way a part of several bytes which together form Config Word.

During programming, the bytes of Config Word are written to the microcontroller's ROM memory and stored in special registers which are not available to the user. On the basis of these bits, the microcontroller "knows" what to do, although it is not explicitly specified in the (written) program.



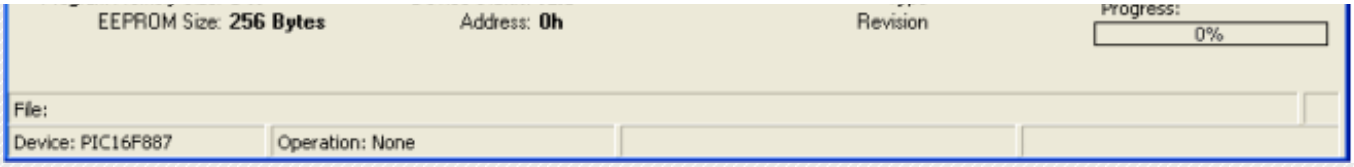


Fig.8-3 PICflash Program

External oscillator in EC mode

The external clock (EC) mode uses the system clock source configured from external oscillator. The frequency of this clock source is unlimited (0- 20MHz).

Fig. 8-4 External Oscillator

This mode has the following advantages:

- The external clock source is connected to the OSC1 input and the OSC2 is available for general purpose I/O;
- It is possible to synchronize the operation of the microcontroller with the rest of on board electronics;
- In this mode the microcontroller starts operating immediately after the power is on. There is no delay required for frequency stabilization; and
- Temporary stopping the external clock input has the effect of halting the device while leaving all data intact. Upon restarting the external clock, the device resumes operation as if nothing has happened.

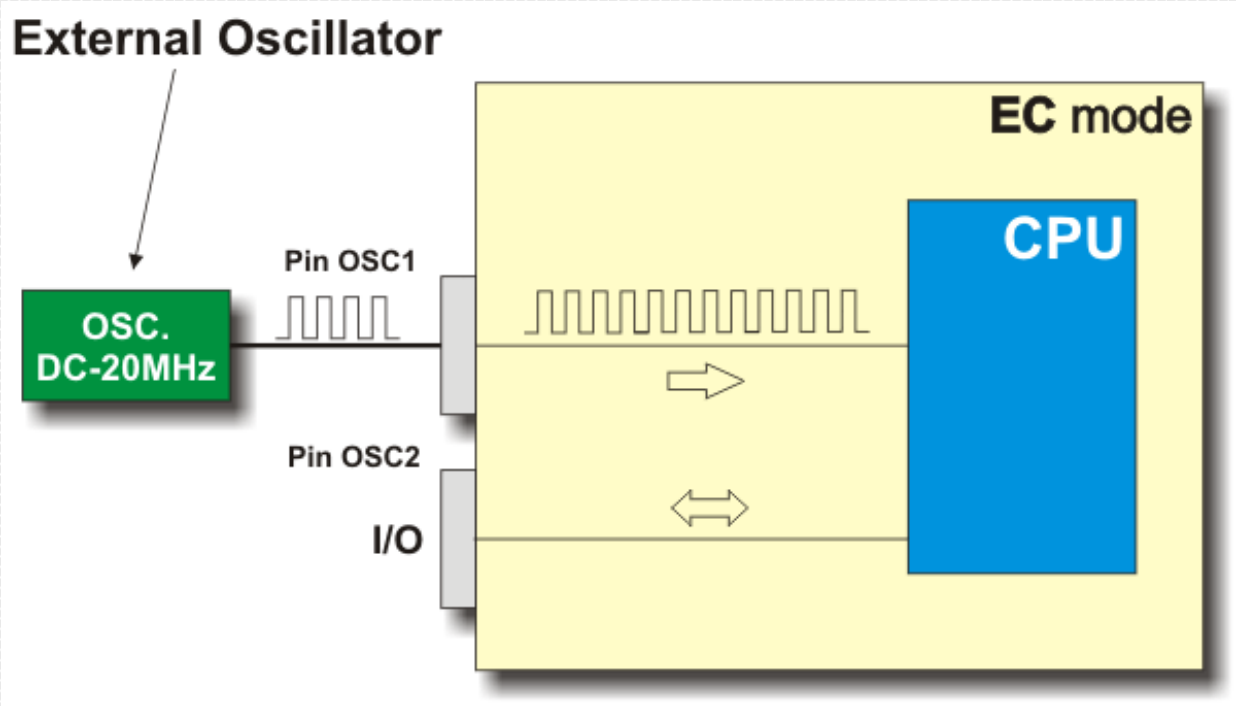


Fig. 8-5 External Oscillator in EC Mode

External oscillator in LP, XT or HS mode

Fig. 8-6 Two Types of External Oscillators

The LP, XT and HS modes support the usage of internal oscillator for configuring clock source. The frequency of this source is determined by quartz crystal or ceramic resonators connected to the OSC1 and OSC2 pins. Depending on features of the component in use, select one of the following modes:

LP mode (Low Power) is used for low-frequency quartz crystal only. This mode is designed to drive only 32.768 kHz crystals usually embedded in quartz watches. It is easy to recognize them by small size and specific cylindrical shape. The current consumption is the least of the three modes;

XT mode is used for intermediate-frequency quartz crystals up to 8 MHz. The current consumption is the medium of the three modes ;and

HS mode (High Speed) is used for high-frequency quartz crystals over 8 MHz. The current consumption is the highest of the three modes.

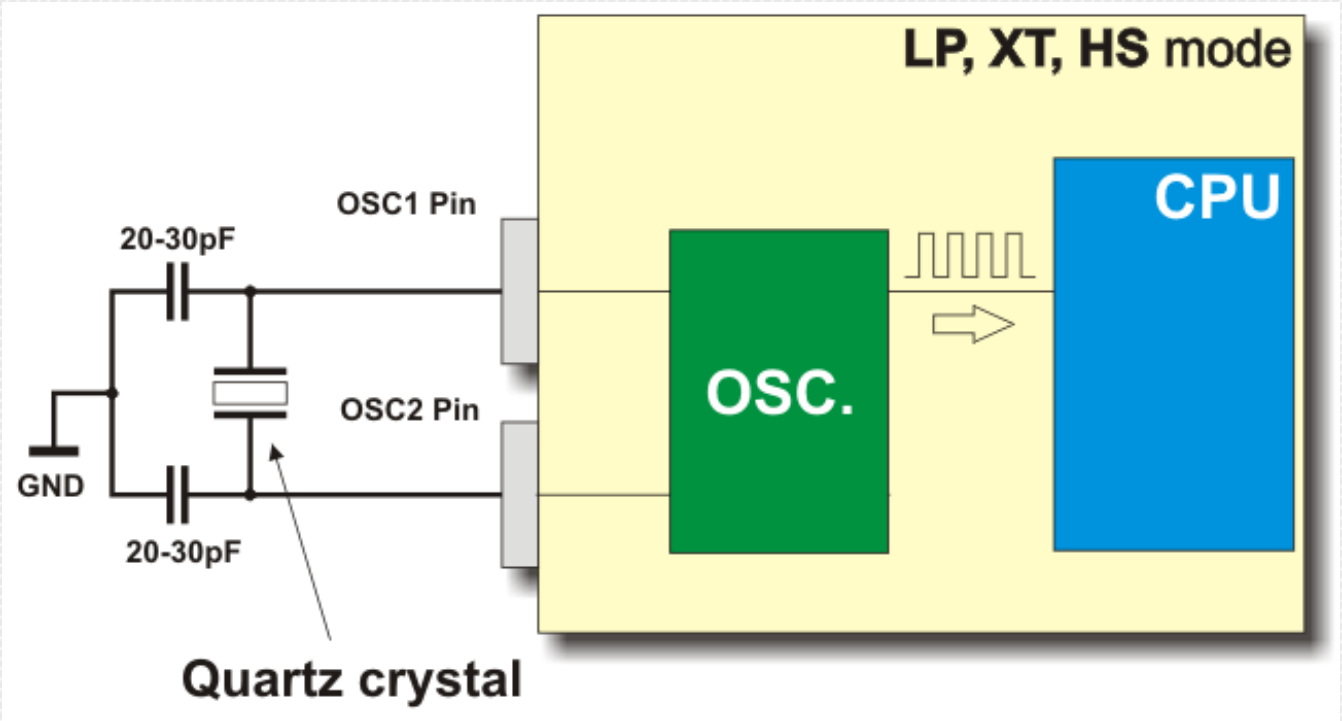


Fig.8-7 Schematic of External Oscillator and Additional External Components

Ceramic resonators in XT or HS mode

Fig.8-8 Ceramic Resonator

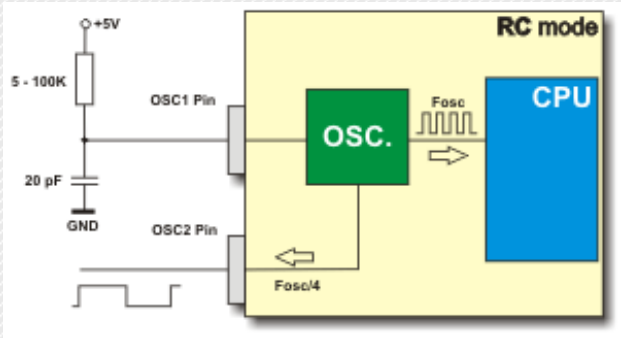
Ceramic resonators are by their features similar to quartz crystals. This is why they are connected in the same way. Unlike quartz crystals, they are cheaper and oscillators containing them have a bit worse characteristics. They are used for clock frequencies ranging between 100 kHz and 20 MHz.

External oscillator in RC and RCIO mode

There are certainly many advantages in using elements for frequency stabilization, but sometimes they are really unnecessary. It is mostly sufficient that the oscillator operates at frequency not precisely defined so that embedding of such expensive elements means a waste of money. The simplest and cheapest solution in these situations is to use one resistor and one capacitor for the operation of oscillator. There are two modes:

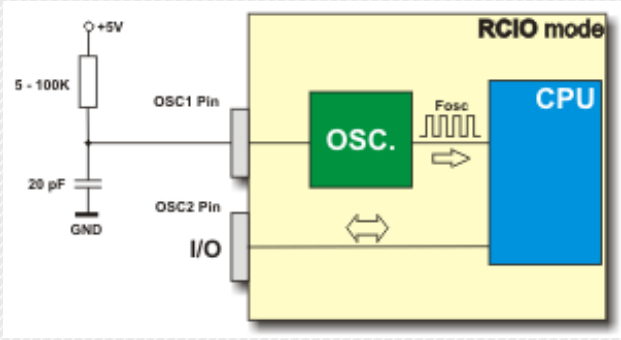
RC mode. In RC mode, the RC circuit is connected to the OSC1 pin as shown in figure. The OSC2 pin outputs the RC oscillator frequency divided by 4. This signal may be used for calibration, synchronization or other application requirements.

Fig. 8-9 RC Mode



RCIO mode. Similar to the previous case, the RC circuit is connected to the OSC1 pin. This time, the available OSC2 pin is used as additional general purpose I/O pin.

Fig. 8-10 RCIO Mode



In both cases, it is recommended to use components as shown in figure.
The frequency of such oscillator is calculated according to the formula f = 1/T in which:
f = frequency [Hz]
T = R*C = time constant [s]
R = resistor resistance [Ω]
C = capacitor capacity [F]

Internal Clock Modes

The internal oscillator circuit consists of two separate oscillators that can be selected as the system clock source:

The **HFINTOSC** oscillator is factory calibrated and operates at 8 MHz. Its frequency can be user-adjusted via software using bits of the OSCTUNE register; and

The **LFINTOSC** oscillator is not factory calibrated and operates at 31kHz.

Similar to the external oscillator, the internal one can also operate in several modes. The mode is selected in the same way as in case of external oscillator- using bits of the Config Word register. In other words, everything is performed within PC software, immediately before program writing to the microcontroller starts.

Internal oscillator in INTOSC mode
In this mode, the OSC1 pin is available as general purpose I/O while the OSC2 pin outputs selected internal oscillator frequency divided by 4.

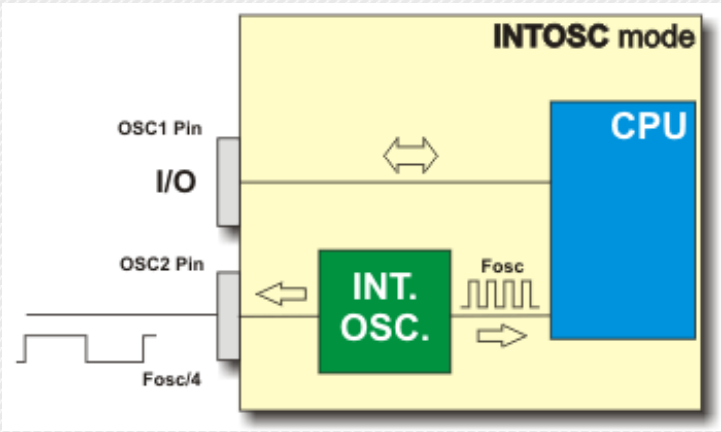


Fig. 8-11 INTOSC Mode

Internal oscillator in INTOSCIO mode
In this mode, both pins are available for general purpose I/O.

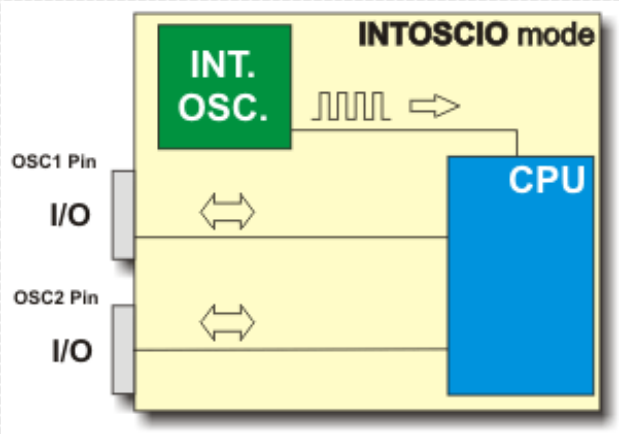


Fig. 8-12 INTOSCIO Mode

Internal Oscillator Settings

The internal oscillator consists of two separate circuits.

1. The high-frequency internal oscillator HFINTOSC is connected to the postscaler (frequency divider). It is factory calibrated and operates at 8MHz. Using postscaler, this oscillator can output clock sources at one of seven frequencies which can be selected via software using the IRCF2, IRCF1 and IRCF0 pins of the OSCCON register.

The HFINTOSC is enabled by selecting one of seven frequencies (between 8 MHz and 125 kHz) and setting the System Clock Source (SCS) bit of the OSCCON register afterwards. As seen in figure below, everything is performed using bits of the OSCCON register.

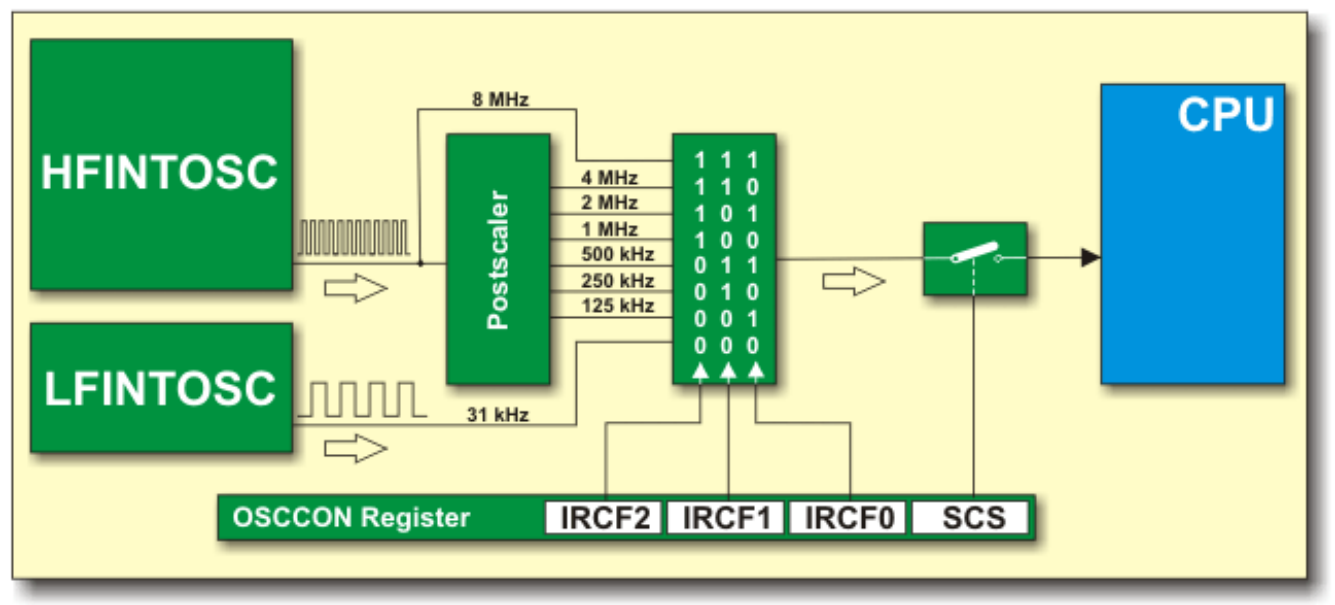


Fig. 8-13 Internal Oscillator settings

2. The low-frequency oscillator LFINTOSC is uncalibrated and operates at 31 kHz. It is enabled by selecting this frequency (bits of the OSCCON register) and setting the SCS bit of the same register.

Two-Speed Clock Start-up Mode

Two-Speed Clock Start-up mode is used to provide additional power savings when the microcontroller operates in sleep mode. What is this all about?

When configured to operate in LP, XT or HS modes, the external oscillator will be switched off on transition to sleep in order to reduce the overall power consumption of the device.

When conditions for wake-up are met, the microcontroller will not immediately start operating because it has to wait for clock signal frequency to become stable. Such delay lasts for exactly 1024 pulses. After that, the microcontroller proceeds with program execution. The problem is that very often only a few instructions are performed before the microcontroller is set up to Sleep mode again. It means that most of time as well as power obtained from batteries is wasted. This problem is solved by using internal oscillator for program execution while these 1024 pulses are counted. Afterwards, as soon as the external oscillator frequency becomes stable, it will automatically take over the “leading role”. The whole process is enabled by setting one bit of the configuration word. In order to program the microcontroller it is necessary to select the Int-Ext Switchover option in software.

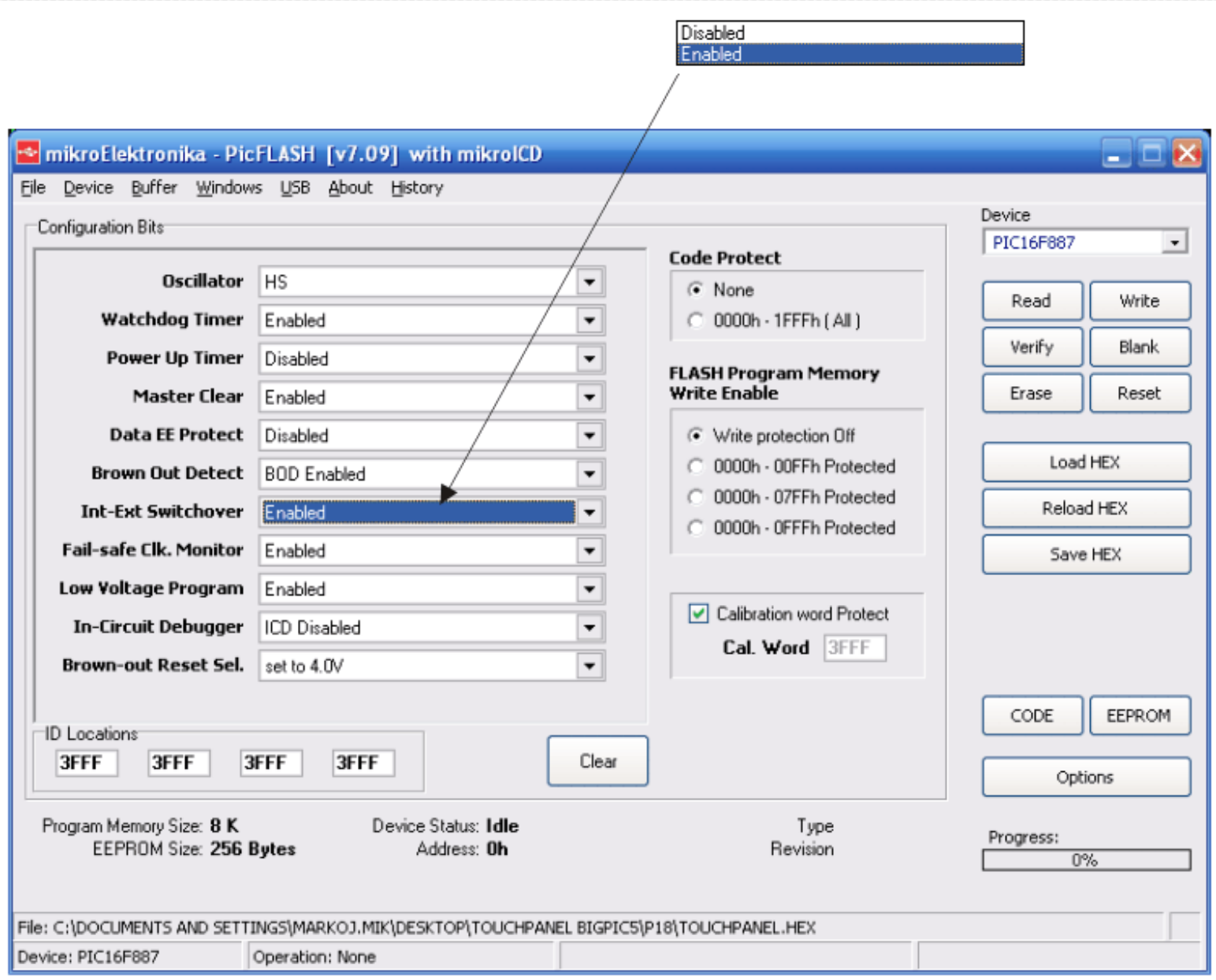


Fig.8-14 Enable Int-Ext Switchover

Fail-Safe Clock Monitor

The Fail-Safe Clock Monitor (FSCM) monitors the operation of external oscillator and allows the microcontroller to proceed with program execution even the external oscillator fails for some reason. In this case, the internal oscillator takes over its role.

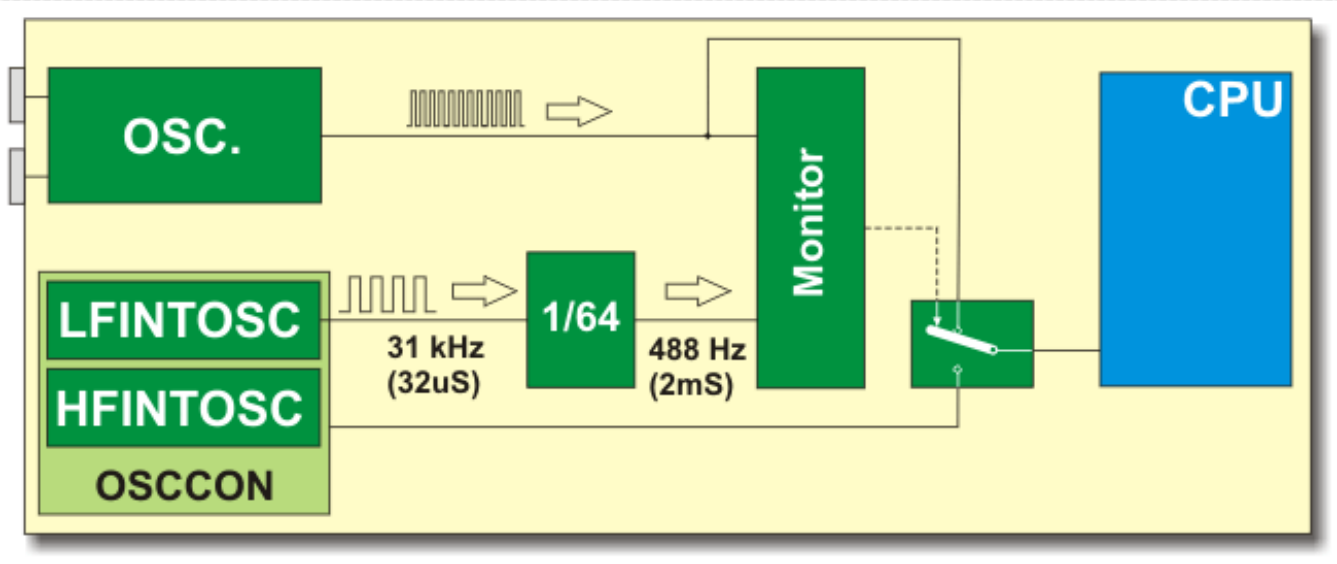
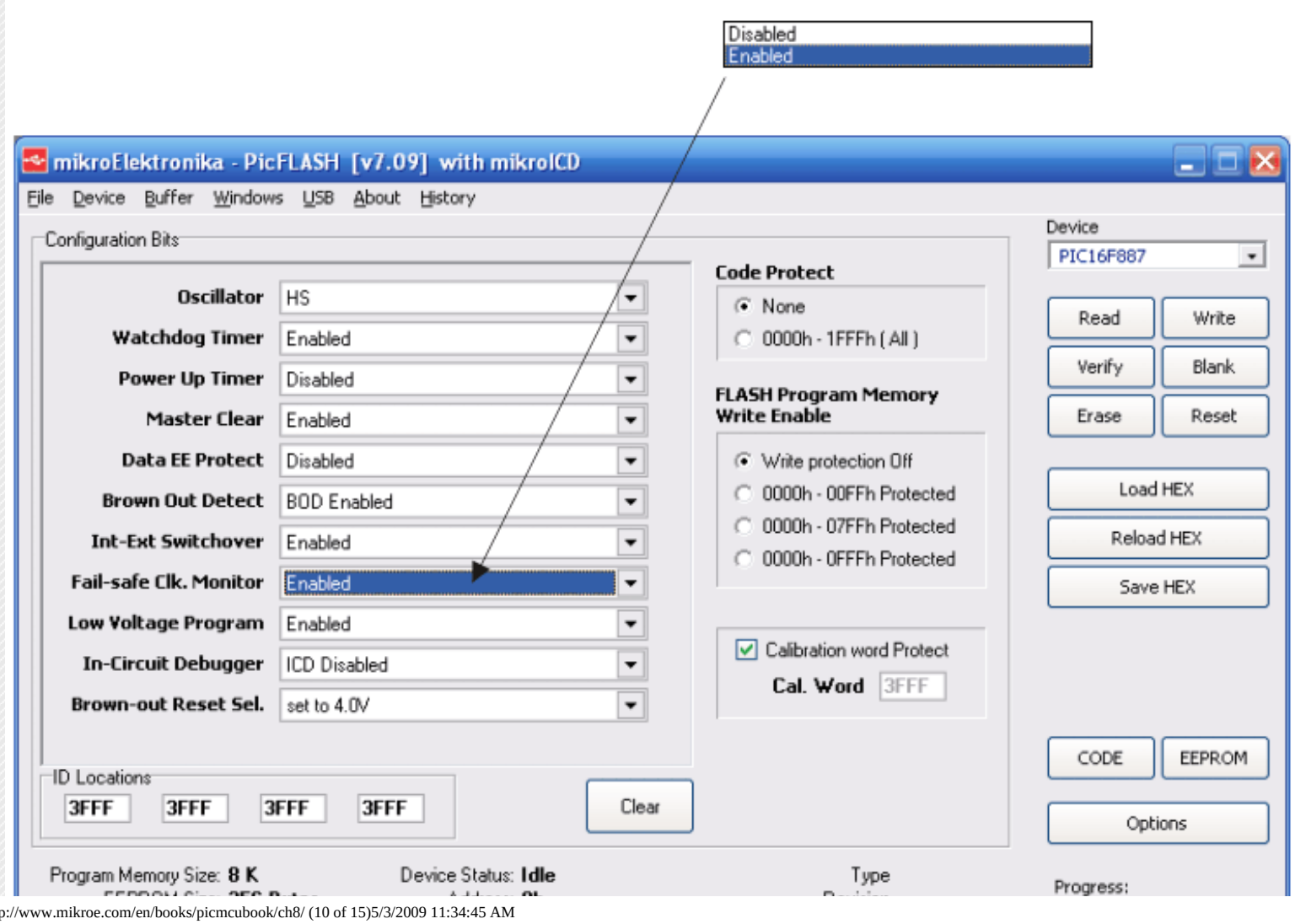


Fig. 8-15 Fail-Safe Clock Monitor

The fail-safe clock monitor detects a failed oscillator by comparing the internal and external clock sources. In case it takes more than 2mS for the external oscillator clock to come, the clock source will be automatically switched. The internal oscillator will thereby continue operating controlled by the bits of the OSCCON register. When the OSFIE bit of the PIE2 register is set, an interrupt will be generated. The system clock will continue to be sourced from internal clock until the device successfully restarts the external oscillator and switches back to external operation.

Similarly to the previous cases, this module is enabled by changing configuration word just before the programming of chip starts. This time, it is done by selecting the Fail-Safe Clk. Monitor option.



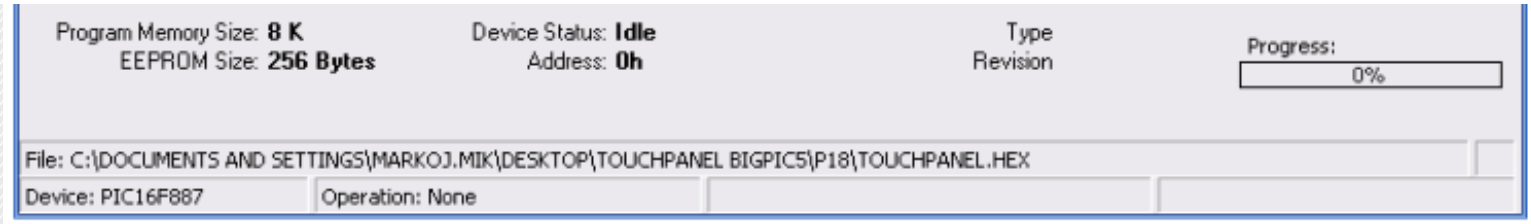


Fig. 8-16 Enabling Fail-Safe Clock Monitor

OSCTUNE Register

Modifications in the OSCTUNE register affect the HFINTOSC frequency, but not the LFINTOSC frequency. Furthermore, there is no indication during operation that shift has occurred.

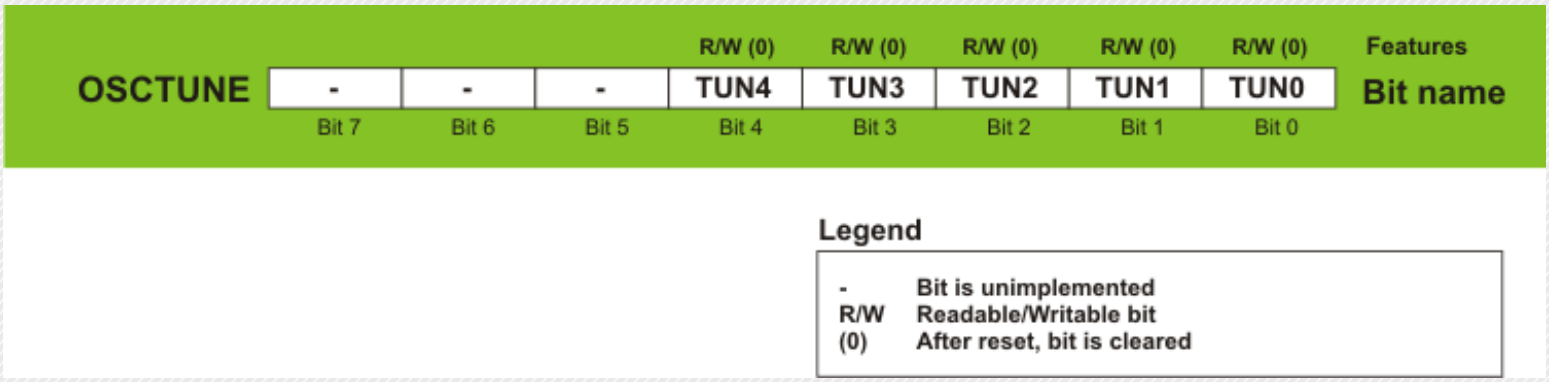


Fig. 8-17 OSCTUNE Register

TUN4 - TUN0 Frequency Tuning bits. By combining these five bits, the 8MHz oscillator frequency shifts. In this way, the frequencies obtained by its division in the postscaler shift too.

TUN4	TUN3	TUN2	TUN1	TUN0	Frequency
0	1	1	1	1	Maximal
0	1	1	1	0	
0	1	1	0	1	
0	0	0	0	1	
0	0	0	0	0	Calibrated
1	1	1	1	1	
1	0	0	1	0	
1	0	0	0	1	
1	0	0	0	0	Minimal

Table 8-2 Frequency Tuning Bits

EEPROM

EEPROM is neither part of program memory (ROM) nor data memory (RAM), but a special memory segment. Even these memory locations are not easily and quickly accessed as other registers, they are of great importance because the EEPROM data are permanently saved (after the power supply goes off). EEPROM data can be also changed at any moment. Because of these exceptional features, each byte of EEPROM is valuable.

The PIC16F887 microcontroller has 256 locations of data EEPROM controlled by the bits of the following registers:

- EECON1 (control register);
- EECON2 (control register);
- EEDAT (saves data ready for write and read); and
- EEADR (saves address of EEPROM location to be accessed).

In addition, EECON2 is not true register, it does not physically exist. It is used in write program sequence only.

The EEDATH and EEADRH registers belong to the same group as the registers used during EEPROM write and read. Both of them are therefore used for program (FLASH) memory write and read.

Since this is considered a risk zone (you surely do not want your microcontroller to accidentally erase your program), we will not discuss it further, but advise you to be careful.

EECON1 Register

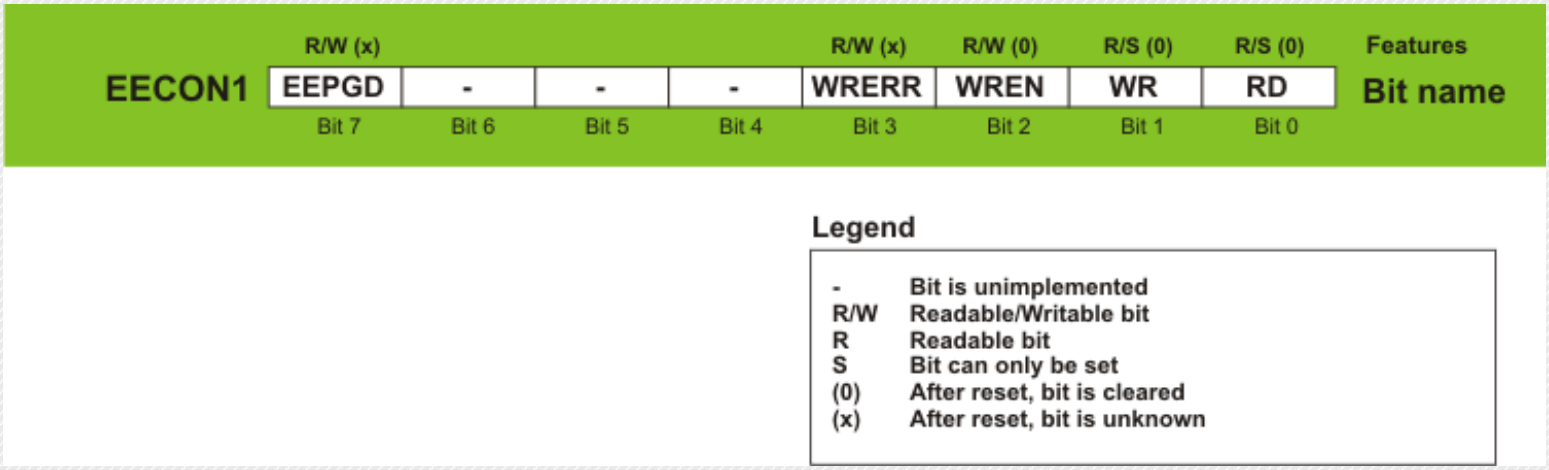


Fig.8-18 EECON1 Register

EEPGD - Program/Data EEPROM Select bit

- 1 - Access program memory; and
- 0 - Access EEPROM memory.

WRERR - EEPROM Error Flag bit

- 1 - Write operation is prematurely terminated and error has occurred; and
- 0 - Access EEPROM memory.

WREN - EEPROM Write Enable bit.

- 1 - Write to data EEPROM enabled; and
- 0 - Write to data EEPROM disabled.

WR - Write Control bit

- 1 - Initiates write to data EEPROM; and
- 0 - Write to data EEPROM is complete.

RD - Read Control bit

- 1 - Initiates read from data EEPROM; and
- 0 - Read from data EEPROM disabled.

Read from EEPROM Memory

In order to read data EEPROM memory, follow the procedure below:

Step 1: Write an address (00h - FFh) to the EEADR register;
Step 2: Select EEPROM memory block by clearing the EEPGD bit of the EECON1 register;
Step 3: To read location, set the RD bit of the same register; and
Step 4: Data is stored in the EEDAT register and ready to use.

The following example illustrates data EEPROM read:

```
BSF   STATUS,RP1   ;
BCF   STATUS,RP0   ; Access bank 2
MOVF  ADDRESS,W    ; Move address to the W register
MOVWF EEADR        ; Write address
BSF   STATUS,RP0   ; Access bank 3
BCF   EECON1,EEPGD ; Select EEPROM
BSF   EECON1,RD    ; Read data
BCF   STATUS,RP0   ; Access bank 2
MOVF  EEDATA,W     ; Data is stored in the W register
```

Write to Data EEPROM Memory

In order to write data to EEPROM memory, first it is necessary to write the address to the EEADR register first and data to the EEDAT register afterwards. Then you have to follow a special sequence to initiate write for each byte. Interrupts must be disabled during this procedure.

Data EEPROM write is illustrated in the example below:

```
BSF   STATUS,RP1
BSF   STATUS,RP0
BTFSC EECON,WR1    ; Wait for the previous write to complete
GOTO  $-1          ;
BCF   STATUS,RP0   ; Bank 2
MOVF  ADDRESS,W    ; Move address to W
MOVWF EEADR        ; Write address
MOVF  DATA,W       ; Move data to W
MOVWF EEDATA       ; Write data
BSF   STATUS,RP0   ; Bank 3
BCF   EECON1,EEPGD ; Select EEPROM
BSF   EECON1,WREN  ; Write to EEPROM enabled
BCF   INCON,GIE    ; All interrupts disabled
MOVLW 55h          ; Required sequence start
MOVWF EECON2
MOVLW AAh
MOVWF EECON2       ; Required sequence end
BSF   EECON1,WR
BSF   INTCON,GIE   ; Interrupts enabled
BCF   EECON1,WREN  ; Write to EEPROM disabled
```

Reset! Black-out, Brown-out or Noises?

On reset, the microcontroller immediately stops operation and clears its registers. Reset signal may be generated externally at any moment (low logic level on the MCLR pin). If needed it can be also generated by internal control logic. Power-on always causes reset. Namely, because of many transitional events which take place when power supply is on (switch contact flashing and sparkling, slow voltage rise, gradual clock frequency stabilization etc.), it is necessary to provide a certain time delay before the microcontroller starts operating. Two internal timers- PWRT and OST are in charge of that. The first one can be enabled or disabled during program writing. The scenario is as follows:

When power supply voltage reaches 1.2 - 1.7V, a circuit called **Power-up** timer resets the microcontroller within approximately 72mS. Immediately upon this time has run out, the reset signal generates another timer called **Oscillator start-up timer** within 1024 quartz oscillator periods. When this delay is over (marked as T reset in figure) and the MCLR pin is set high, the microcontroller starts to execute the first instruction in the program.

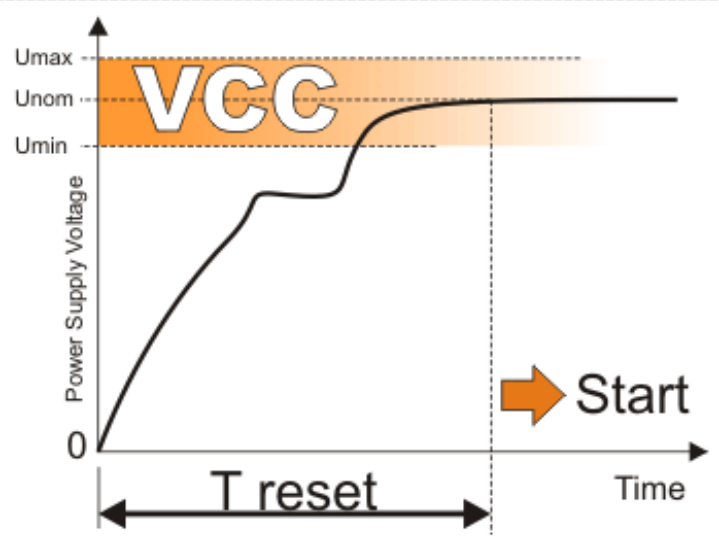


Fig. 8-19 Oscillator Start-Up Time Delay

Apart from such- "controlled" reset which occurs at the moment power goes on, there are another two resets called **Black-out** and **Brown-out** which may occur during operation as well as at the moment power goes off.

Black-out reset

Black-out reset takes place when the power supply normally goes off. In that case, the microcontroller has no time to do anything unpredictable simply because the voltage drops very fast beneath its minimal value. In other words- the light goes off, curtain falls down and the show is over!

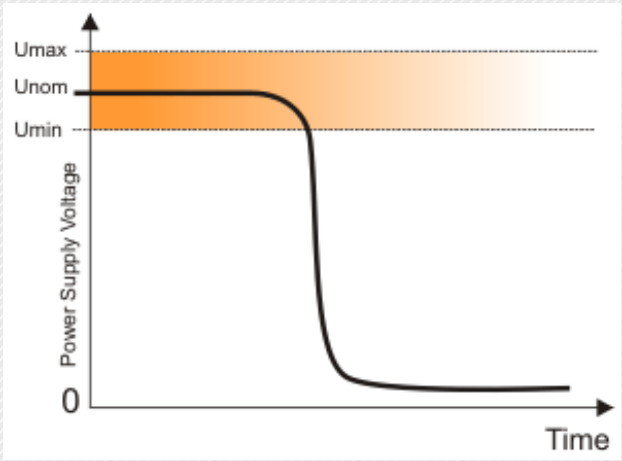


Fig. 8-20 Black-Out Reset at Loss Of Power

Brown-out reset

When power supply voltage drops slowly (typical example of that is battery discharge although the microcontroller experiences far faster voltage drop as a slow process), the internal electronics gradually stops operating and so called Brown-out reset occurs. In that case, prior to the microcontroller stops operating there is a serious danger that circuits which operate at higher voltages start perform unpredictable. It can also causes fatal changes in the program itself because it is saved in on-chip flash memory.

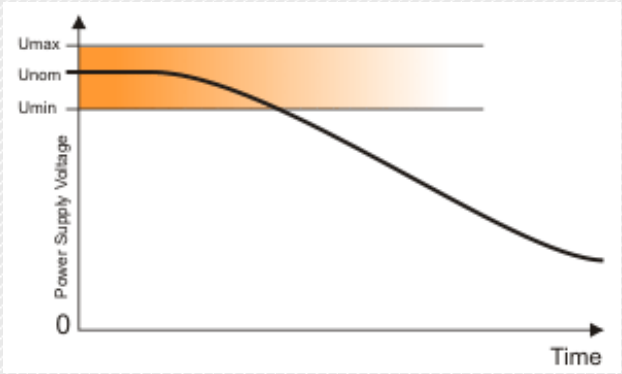


Fig. 8-21 Brown-Out Reset at Gradual Loss Of Power

Noises

This is a special kind of Brown-out reset which occurs in industrial environment when the power supply voltage “blinks” for a moment and drops its value beneath minimal level. Even short, such noise in power line may catastrophically affect the operation of device.

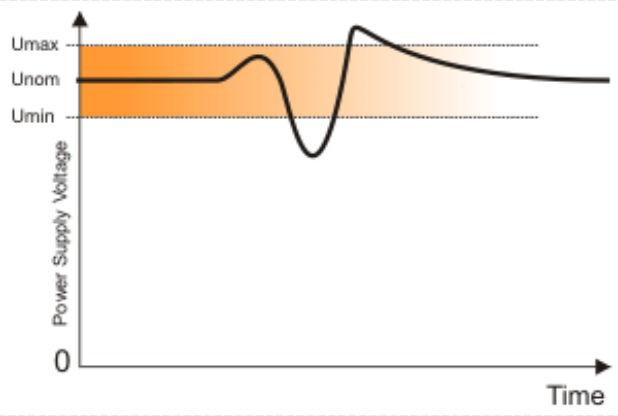


Fig. 8-22 Noises

MCLR pin

Logic zero (0) on the MCLR pin causes immediate and regular reset. It is recommended to be connected as shown in figure below. The function of additional components is to sustain “pure” logic one (1) during normal operation. If their values are such to provide high logic level on the pin only upon T reset is over, the microcontroller will immediately start operating. This feature may be very useful when it is necessary to synchronize the operation of the microcontroller with additional electronics or the operation of several microcontrollers.

In order to avoid any error which may occur on Brown-out reset, the PIC 16F887 has built in ‘defense mechanism’. It is a simple but effective circuit which reacts every time the voltage power supply drops below 4V and holds that level for more than 100 micro seconds. In that case, this circuit generates reset signal and since that moment the whole microcontroller operates as if it has just been switched on.

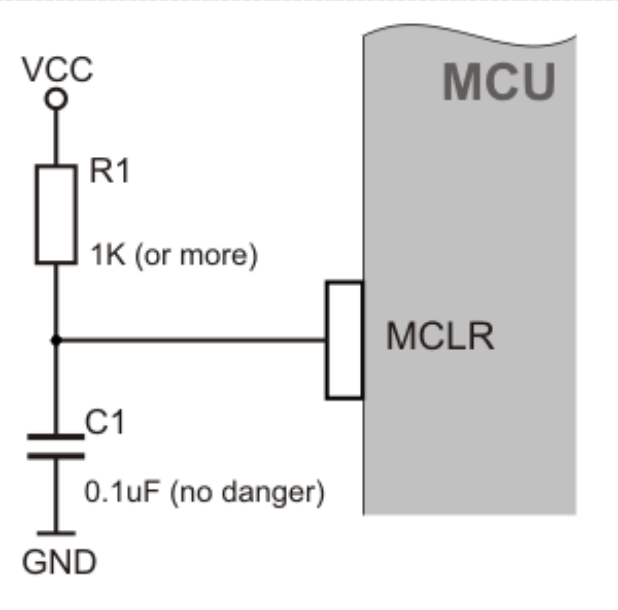


Fig. 8-23 Master Clear Pin

Previous Chapter | Table of Contents | Next Chapter

Chapter 9: Instruction Set

It has been already mentioned that microcontrollers differs from other integrated circuits. Most of them are ready for installation into the target device just as they are, this is not the case with the microcontrollers. In order that the microcontroller may operate, it needs precise instructions on what to do. In other words, a program that the microcontroller should execute must be written and loaded into the microcontroller. This chapter covers the commands which the microcontroller "understands". The instruction set for the 16FXX includes 35 instructions in total. Such a small number of instructions is specific to the RISC microcontroller because they are well-optimized from the aspect of operating speed, simplicity in architecture and code compactness. The only disadvantage of RISC architecture is that the programmer is expected to cope with these instructions.

Instruction	Description	Operation	Flag	CLK	*
Data Transfer Instructions					
MOVLW k	Move constant to W	k -> w		1	
MOVWF f	Move W to f	W -> f		1	
MOVF f,d	Move f to d	f -> d	Z	1	1, 2
CLRW	Clear W	0 -> W	Z	1	
CLRF f	Clear f	0 -> f	Z	1	2
SWAPF f,d	Swap nibbles in f	f(7:4),(3:0) -> f(3:0), (7:4)		1	1, 2

Arithmetic-logic Instructions					
ADDLW k	Add W and constant	W+k -> W	C, DC, Z	1	
ADDWF f,d	Add W and f	W+f -> d	C, DC ,Z	1	1, 2
SUBLW k	Subtract W from constant	k-W -> W	C, DC, Z	1	
SUBWF f,d	Subtract W from f	f-W -> d	C, DC, Z	1	1, 2
ANDLW k	Logical AND with W with constant	W AND k -> W	Z	1	
ANDWF f,d	Logical AND with W with f	W AND f -> d	Z	1	1, 2
ANDWF f,d	Logical AND with W with f	W AND f -> d	Z	1	1, 2
IORLW k	Logical OR with W with constant	W OR k -> W	Z	1	
IORWF f,d	Logical OR with W with f	W OR f -> d	Z	1	1, 2
XORWF f,d	Logical exclusive OR with W with constant	W XOR k -> W	Z	1	1, 2
XORLW k	Logical exclusive OR with W with f	W XOR f -> d	Z	1	
INCF f,d	Increment f by 1	f+1 -> f	Z	1	1, 2
DECF f,d	Decrement f by 1	f-1 -> f	Z	1	1, 2
RLF f,d	Rotate left f through CARRY bit		C	1	1, 2
RRF f,d	Rotate right f through CARRY bit		C	1	1, 2
COMF f,d	Complement f	f -> d	Z	1	1, 2
Bit-oriented Instructions					
BCF f,b	Clear bit b in f	0 -> f(b)		1	1, 2
BSF f,b	Clear bit b in f	1 -> f(b)		1	1, 2
Program Control Instructions					
BTFSC f,b	Test bit b of f. Skip the following instruction if clear.	Skip if f(b) = 0		1 (2)	3
BTFSS f,b	Test bit b of f. Skip the following instruction if set.	Skip if f(b) = 1		1 (2)	3
DECFSZ f,d	Decrement f. Skip the following instruction if clear.	f-1 -> d skip if Z = 1		1 (2)	1, 2, 3
INCFSZ f,d	Increment f. Skip the following instruction if set.	f+1 -> d skip if Z = 0		1 (2)	1, 2, 3
GOTO k	Go to address	k -> PC		2	
CALL k	Call subroutine	PC -> TOS, k -> PC		2	
RETURN	Return from subroutine	TOS -> PC		2	
RETLW k	Return with constant in W	k -> W, TOS -> PC		2	

RETFIE	Return from interrupt	TOS -> PC, 1 -> GIE		2
Other instructions				
NOP	No operation	TOS -> PC, 1 -> GIE		1
CLRWDT	Clear watchdog timer	0 -> WDT, 1 -> TO, 1 -> PD	TO, PD	1
SLEEP	Go into sleep mode	0 -> WDT, 1 -> TO, 0 -> PD	TO, PD	1

Table 9-1 16Fxx Instruction Set

*1 When an I/O register is modified as a function of itself, the value used will be that value present on the pins themselves.
*2 If the instruction is executed on the TMR register and if d=1, the prescaler will be cleared.
*3 If the PC is modified or test result is logic one (1), the instruction requires two cycles.

Data Transfer Instructions

Data Transfer within the microcontroller takes place between working register W (called accumulator) and a register which represents any location of internal RAM regardless of whether it is about special function or general purpose registers.

First three instructions move literal to W register (MOVLW stands for **move Literal to W**), move data from W register to RAM and from RAM to W register (or to the same RAM location with change on flag Z only). Instruction CLRF clears f register, whereas CLRW clears W register. SWAPF instruction swaps nibbles within f register (one nibble contains four bits).

Arithmetic-logic Instructions

Similar to most microcontrollers, PIC supports only two arithmetic instructions- addition and subtraction. Flags C, DC, Z are automatically set depending on the results of addition or subtraction. The only exception is the flag C. Since subtraction is performed as addition with negative value, the flag C is inverted after subtraction. It means that the flag C is set if it is possible to perform operation and cleared if the larger number is subtracted from smaller one. Logic one (1) of the PIC is able to perform operations AND, OR, EX-OR, inverting (COMF) and rotation (RLF and RRF).

Instructions which rotate a register actually rotate its bits through the flag C by one bit left (toward bit 7) or right (toward bit 0). The bit shifted from the register is moved to the flag C which is automatically moved to the bit on the opposite side of the register.

Bit-oriented Instructions

Instructions BCF and BSF clear or set any bit in memory. Although it seems to be a simple operation, it is not like that. CPU first reads the entire byte, changes one its bit and rewrites the whole byte to the same location.

Program Control Instructions

The PIC16F887 executes instructions GOTO, CALL, RETURN in the same way as all other microcontrollers do. A difference is that stack is independent from internal RAM and has 8 levels. The 'RETLW k' instruction is identical to RETURN instruction, with exception that a constant defined by instruction operand is written to the W register prior to return from subroutine. This instruction enables **Lookup** tables to be easily created by creating a table as a subroutine consisting of 'RETLWk' instructions, where the literals 'k' belong to the table. The next step is to write the position of the literals k (0, 1, 2, 3...n) to W register and call the subroutine (table) using the CALL instruction. Table below consists of the following literals: k0, k1, k2...kn.

```
Main   movlw 2     ;write number 2 to accumulator
call   Lookup      ;jump to the lookup table
Lookup addwf PCL,f ;add accumulator and program cur
                   ;rent address (PCL)
retlw  k0          ;return from subroutine (accumulator contains
k0)
retlw  k1          ;...
retlw  k2          ;...
...                ;...
...                ;...
retlw  kn          ;return from subroutine (accumulator contains
kn)
```

The first line of the subroutine (instruction `ADDWF PCL,f`)simply adds a literal "k" from W register and table start address which is stored in the PCL register. The result is real data address in program memory. Upon return from the subroutine, the W register will contain the addressed literal k. In this case, it is the "k2" literal.

RETFIE (**RETurn From IntErrupt**) represents a return from interrupt routine. In contrast to the RETURN instruction, it may automatically set the GIE bit (**Global Interrupt Enable**). When an interrupt occurs this bit is automatically cleared. Only the program counter is pushed to the stack, which means that there is no auto save of registers' status and the current status either. The problem is solved by saving status of all important registers at the beginning of interrupt routine. These values are retrieved to these registers immediately before leaving the interrupt routine.

Conditional jumps are executed by two instructions: BTFSC and BTFSS. Depending on the state of bit being tested in the 'f' register, the following instruction will be skipped or not.

Instruction Execution Time

All instructions are single-cycle instructions. The only exception may be conditional branch instructions (if condition is met) or instructions being executed upon the program counter. In both cases, two cycles are required for instruction execution where the second cycle is executed as a NOP (**No Operation**). A single-cycle instruction consists of four clock cycles. If 4MHz oscillator is used, a nominal time for instruction execution is 1 µS. In case of jump, the instruction execution time is 2 µS.

Legend

f - Any memory location (register);
W - Working register (accumulator);
b - Bit address within an 8-bit register;
d - Destination bit;
[label] - Set of 8 characters indicating start of particular address in the program;
TOS - Top of stack;
[] - Option;
<> - bit field in register (several bit addresses);
C - Carry/Borrow bit of the STATUS register;
DC - Digit Carry bit of the STATUS register; and
Z - Zero bit of the STATUS register.

ADDLW - Add literal and W

Syntax: [label] ADDLW k

Description: The content of the register **W** is added to the 8-bit literal **k**. The result is stored in the **W** register.

Operation: (**W**) + **k** -> **W**

Operand: 0 ≤ **k** ≤ 255

Status affected: C, DC, Z

Number of cycles: 1

EXAMPLE:

```
....
[label] ADDLW 0x15
```

```
Before instruction execution: W=0x10
After instruction: W=0x25
C=0 (the result is not greater than 0xFF, which means that Carry
has not occurred).
```

ADDWF - Add W and f

Syntax: [label] ADDWF f, d

Description: Add the contents of the **W** and **f** registers.

If **d = w** or **d = 0** the result is stored in the **W** register.
If **d = f** or **d = 1** the result is stored in register **f**.

Operation: (**W**) + (**f**) -> **d**

Operand: 0 ≤ **f** ≤ 127, d [0,1]

Status affected: C, DC, Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] ADDWF REG,w
```

```
Before instruction execution: W = 0x17
REG = 0xC2
After instruction: W = 0xD9
REG = 0xC2
C=0 (No carry occurs, i.e. the result is maximum 8-bit long).
```

EXAMPLE 2:

```
....
[label] ADDWF INDF,f
```

```
Before instruction execution: W=0x17
FSR = 0xC2 Register at address 0xC2 contains the value 0x20
After instruction: W = 0x17
FSR=0xC2, Register at address 0xC2 contains the value 0x37
```

ANDLW - AND literal with W

Syntax: [label] ANDLW k

Description: The content of the register **W** is AND'ed with the 8-bit literal **k**. It means that the result will contain one (1) only if both corresponding bits of operand are ones (1). The result is stored in the **W** register.

Operation: (**W**) AND **k** -> **W**

Operand: 0 ≤ **k** ≤ 255

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] ANDLW 0x5F
```

```
Before instruction execution: W = 0xA3 ; 1010 0011 (0xA3)
                              ; 0101 1111 (0x5F)
                              ------------------
After instruction:  W = 0x03 ; 0000 0011 (0x03)
Z = 0 (result is not 0)
```

EXAMPLE 2:

```
....
[label] ANDLW 0x55
```

```
Before instruction execution: W = 0xAA ; 1010 1010 (0xAA)
                              ; 0101 0101 (0x55)
                              ------------------
After instruction:  W = 0x00 ; 0000 0000 (0x00)
Z = 1( result is 0)
```

ANDWF - AND W with f

Syntax: [label] ANDWF f,d

Description: AND the **W** register with register **f**.
If **d = w** or **d = 0**, the result is stored in the **W** register.
If **d = f** or **d = 1**, the result is stored in register **f**.

Operation: (**W**) AND (**f**) -> **d**

Operand: 0 ≤ **f** ≤ 127, d[0,1]

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] ANDWF REG,f
```

```
Before instruction execution: W = 0x17, REG = 0xC2 ; 0001 0111
(0x17)
                                                   ; 1100 0010 (0xC2)
                                                   ------------------
After instruction: W = 0x17, REG = 0x02  ; 0000 0010 (0x02)
```

EXAMPLE 2:

```
....
[label] ANDWF FSR,w
```

```
Before instruction execution: W = 0x17, FSR = 0xC2 ; 0001 0111
(0x17)
                                                   ; 1100 0010 (0xC2)
                                                   ------------------
After instruction: W = 0x02, FSR = 0xC2  ; 0000 0010 (0x02)
```

BCF - Bit Clear f

Syntax: [label] BCF f, b

Description: Bit **b** of register **f** is cleared.

Operation: (0) -> f(b)

Operand: 0 ≤ **f** ≤ 127, 0 ≤ **b** ≤ 7

Status affected: -

Number of cycles: 1

EXAMPLE 1:

```
....
[label] BCF REG,7
```

```
Before instruction execution: REG = 0xC7 ; 1100 0111 (0xC7)
After instruction:  REG = 0x47 ; 0100 0111 (0x47)
```

EXAMPLE 2:

```
....
[label] BCF INDF,3
```

```
Before instruction execution: W = 0x17
                    FSR = 0xC2
                    Register at address (FSR)contains the value
0x2F
After instruction:  W = 0x17
                    FSR = 0xC2
                    Register at address (FSR)contains the value
0x27
```

BSF - Bit set f

Syntax: [label] BSF f,b

Description: Bit **b** of register **f** is set.

Operation: 1 -> f (b)

Operand: 0 ≤ **f** ≤ 127, 0 ≤ **b** ≤ 7

Status affected: -

Number of cycles: 1

EXAMPLE 1:

```
....
[label] BSF REG,7
```

```
Before instruction execution: REG = 0x07 ; 0000 0111 (0x07)
After instruction:  REG = 0x87 ; 1000 0111 (0x87)
```

EXAMPLE 2:

```
....
[label] BSF INDF,3
```

```
Before instruction execution: W = 0x17
                    FSR = 0xC2
                    Register at address (FSR)contains the value
0x20
After instruction:  W = 0x17
                    FSR = 0xC2
                    Register at address (FSR)contains the value
0x28
```

BTFSC - Bit test f, Skip if Clear

Syntax: [label] BTFSC f, b

Description: If bit **b** of register **f** is 0, the next instruction is discarded and a NOP is executed instead, making this a two-cycle instruction.

Operation: Discard the next instruction if f(b) = 0

Operand: 0 ≤ **f** ≤ 127, 0 ≤ **b** ≤ 7

Status affected: -

Number of cycles: 1 or 2 depending on bit **b**

EXAMPLE:

```
      ....
LAB_01 BTFSC REG,1 ; Test bit 1 of REG
LAB_02 ....        ; Skip this line if bit = 1
LAB_03 ....        ; Jump here if bit = 0
```

```
Before instruction execution: The program counter was at address
LAB_01.
After instruction:
- if bit 1 of REG is cleared, program counter points to address
LAB_03.
- if bit 1 of REG is set, program counter points to address
LAB_02.
```

BTFSS - Bit test f, Skip if Set

Syntax: [label] BTFSS f, b

Description: If bit **b** of register **f** is 1, the next instruction is discarded and a NOP is executed instead, making this a two-cycle instruction.

Operation: Discard the next instruction if f(b) = 1

Operand: 0 ≤ **f** ≤ 127, 0 ≤ **b** ≤ 7

Status affected: -

Number of cycles: 1 or 2 depending on bit **b**

EXAMPLE:

```
      ....
LAB_01 BTFSS REG,3 ; Test bit 3 of REG
LAB_02 ....        ; Skip this line if bit = 0
LAB_03 ....        ; Jump here if bit = 1
```

```
Before instruction execution: The program counter was at address
```

```
LAB_01
After instruction:
- if bit 3 of REG is cleared, program counter points to address
LAB_03.
- if bit 3 of REG is cleared, program counter points to address
LAB_02.
```

CALL - Calls Subroutine

Syntax: [label] CALL k

Description: Calls subroutine. First the address of the next instruction to execute is pushed onto the stack. It is the PC+1 address. Afterwards, the subroutine address is written to the program counter.

Operation: (PC) + 1 -> (Top Of Stack - TOS)
k -> PC (10 : 0), (PCLATH (4 : 3)) -> PC (12 : 11)

Operand: 0 ≤ **k** ≤ 2047

Flag: -

Status affected: 2

EXAMPLE:

```
        ....
LAB_01 CALL LAB_02 ; Call subroutine LAB_02
        ....
        ....
LAB_02 ....
```

```
Before instruction execution: PC = address LAB_01
                      TOS (top of stack) = x
After instruction:    PC = address LAB_02
                      TOS (top of stack) = LAB_01
```

CLRF - Clear f

Syntax: [label] CLRF **f**

Description: The content of register **f** is cleared and the Z flag of the STATUS register is set.

Operation: 0 -> f

Operand: 0 ≤ **f** ≤ 127

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] CLRF TRISB
```

```
Before instruction execution: TRISB=0xFF
After instruction:  TRISB=0x00
Z = 1
```

EXAMPLE 2:

```
Before instruction execution: FSR=0xC2
                    Register at address 0xC2 contains the value
0x33
After instruction:  FSR=0xC2
                    Register at address 0xC2 contains the value
0x00
                    Z = 1
```

CLRW - Clear W

Syntax: [label] CLRW

Description: Register **W** is cleared and the Z flag of the STATUS register is set.

Operation: 0 -> W

Operand: -

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] CLRW
```

```
Before instruction: W=0x55
After instruction:  W=0x00
                    Z = 1
```

CLRWDT - Clear Watchdog Timer

Syntax: [label] CLRWDT

Description: Resets the **watchdog** timer and the WDT prescaler. Status bits TO and PD are set.

Operation: 0 -> WDT 0 -> WDT prescaler 1 -> TO 1 -> PD

Operand: -

Status affected: TO, PD

Number of cycles: 1

EXAMPLE :

```
....
[label] CLRWDT
```

```
Before instruction execution: WDT counter = x
                    WDT prescaler = 1:128
After instruction:  WDT counter = 0x00
                    WDT prescaler = 0
                    TO = 1
                    PD = 1
                    WDT prescaler = 1: 128
```

COMF - Complement f

Syntax: [label] COMF f, d

Description: The content of register **f** is complemented (logic zeros (0) are replaced by ones (1) and vice versa). If **d** = **w** or **d** = 0 the result is stored in **W**. If **d** = **f** or **d** = 1 the result is stored in register **f**.

Operation: (f) -> d

Operand: 0 ≤ **f** ≤ 127, d[0,1]

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] COMF REG,w
```

```
Before instruction execution: REG = 0x13 ; 0001 0011 (0x13)
                               ; complementing
                               ------------------
After instruction:  REG = 0x13 ; 1110 1100 (0xEC)
                    W = 0xEC
```

EXAMPLE 2:

```
....
[label] COMF INDF, f
```

```
Before instruction execution: FSR = 0xC2
                    Register at address (FSR)contains the value
0xAA
After instruction:  FSR = 0xC2
                    Register at address (FSR)contains the value
0x55
```

DECF - Decrement f

Syntax: [label] DECF f, d

Description: Decrement register **f** by one. If **d** = **w** or **d** = 0, the result is stored in the **W** register. If **d** = **f** or **d** = 1, the result is stored in register **f**.

Operation: (f) - 1 -> d

Operand: 0 ≤ **f** ≤ 127, d[0,1]

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] DECF REG,f
```

```
Before instruction execution: REG = 0x01
                              Z = 0
After instruction:  REG = 0x00
                    Z = 1
```

EXAMPLE 2:

```
....
[label] DECF REG,w
```

```
Before instruction execution: REG = 0x13
                    W = x, Z = 0
After instruction:  REG = 0x13
                    W = 0x12, Z = 0
```

DECFSZ - Decrement f, Skip if 0

Syntax: [label] DECFSZ f, d

Description: Decrement register f by one. If **d** = **w** or **d** = 0, the result is stored in the **W**

register. If **d** = **f** or **d** = **1**, the result is stored in register **f**. If the result is 0, then a NOP is executed instead, making this a two-cycle instruction.

Operation: (f) - 1 -> d

Operand: 0 ≤ **f** ≤ 127, d[0,1]

Status affected: -

Number of cycles: 1 or 2 depending on the result.

EXAMPLE 1:

```
        ....
        MOVLW  .10
        MOVWF  CNT        ;10 -> CNT
Loop    ......
        ......            ;Instruction block
        ......
        DECFSZ CNT,f      ; decrement REG by one
        GOTO   Loop       ; Skip this line if = 0
LAB_03  .......           ; Jump here if = 0
```

In this example, instruction block is executed as many times as the initial value of the variable CNT is, which in this example is 10.

GOTO - Unconditional Branch

Syntax: [label] GOTO k

Description: Unconditional jump to the address **k**.

Operation: (k) -> PC(10:0), (PCLATH(4:3)) -> PC(12:11)

Operand: 0 ≤ **k** ≤ 2047

Status affected: -

Number of cycles: 2

EXAMPLE :

```
        ....
LAB_00  GOTO LAB_01 ; Jump to LAB_01
```

```
        .....
        .....
LAB_01 .....       ; Program continues from here
```

```
Before instruction execution: PC = LAB_00 address
After instruction:  PC = LAB_01 address
```

INCF - Increment f

Syntax: [label] INCF **f, d**

Description: Increment register **f** by one.
If **d** = **w** or **d** = 0, the result is stored in register **W**.
If **d** = **f** or **d** = **1**, the result is stored in register **f**.

Operation: (**f**) + 1 -> d

Operand: 0 ≤ **f** ≤ 127, d[0,1]

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] INCF REG,w
```

```
Before instruction execution: REG = 0x10
                      W = x, Z = 0
After instruction:  REG = 0x10
                      W = 0x11, Z = 0
```

EXAMPLE 2:

```
....
[label] INCF REG,f
```

```
Before instruction execution: REG = 0xFF
                              Z = 0
After instruction:  REG = 0x00
                              Z = 1
```

INCFSZ - Increment f, Skip if 0

Syntax: [label] INCFSZ **f**, **d**

Description: Register **f** is incremented by one. If **d** = **w** or **d** = 0, the result is stored in register **W**. If **d** = **f** or **d** = 1, the result is stored in register **f**. If the result is 0, then a NOP is executed instead, making this a two-cycle instruction.

Operation: (f) + 1 -> d

Operand: 0 ≤ **f** ≤ 127, d[0,1]

Status affected: -

Number of cycles: 1 or 2 depending on the result.

EXAMPLE :

```
      ....
LAB_01 INCFSZ REG,f ; Increment REG by one
LAB_02 .......      ; Skip this line if result is 0
LAB_03 .......      ; Jump here if result is 0
```

The content of program counter Before instruction execution, PC= LAB_01address.

The content of REG after instruction, REG = REG+1. If REG=0, the program counter points to the address of label LAB_03. Otherwise, the program counter points to address of the next instruction, i.e. to LAB_02 address.

IORLW - Inclusive OR literal with W

Syntax: [label] IORLW **k**

Description: The content of the **W** register is OR'ed with the 8-bit literal **k**. The result is stored in register **W**.

Operation: (W) OR (k) -> W

Operand: 0 ≤ **k** ≤ 255

Status affected: -

Number of cycles: 1

EXAMPLE :

```
....
[label] IORLW 0x35
```

```
Before instruction execution: W = 0x9A
After instruction:  W = 0xBF
                    Z = 0
```

IORWF - Inclusive OR W with f

Syntax: [label] IORWF **f**, **d**

Description: The content of register **f** is OR'ed with the content of **W** register. If **d** = **w** or **d** = **0**, the result is stored in the **W** register. If **d** = **f** or **d** = 1, the result is stored in register **f**.

Operation: (**W**) OR (**f**) -> **d**

Operand: 0 ≤ **f** ≤ 127, **d** -> [0,1]

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] IORWF REG,w
```

```
Before instruction execution: REG = 0x13,
                    W = 0x91
After instruction:  REG = 0x13,
                    W = 0x93 Z = 0
```

EXAMPLE 2:

```
....
[label] IORWF REG,f
```

```
Before instruction execution: REG = 0x13,
                              W = 0x91
After instruction:  REG = 0x93,
                    W = 0x91 Z = 0
```

MOVF - Move f

Syntax: [label] MOVF **f**, **d**

Description: The content of register **f** is moved to a destination determined by the operand d. If **d** = **w** or **d** = 0, the content is moved to register **W**. If **d** = **f** or **d** = 1, the content remains in register **f**. Option **d** = 1 is used to test the content of register **f** because this instruction affects the Z flag of the STATUS register.

Operation: (f) -> d

Operand: 0 ≤ **f** ≤ 127, **d** -> [0,1]

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] MOVF FSR,w
```

```
Before instruction execution: FSR=0xC2
                              W=0x00
After instruction:  W=0xC2
                    Z = 0
```

EXAMPLE 2:

```
....
[label] MOVF INDF,f
```

```
Before instruction execution: W=0x17
                      FSR=0xC2, register at address 0xC2 contains
the value 0x00
After instruction:  W=0x17
                      FSR=0xC2, register at address 0xC2 contains
the value 0x00,
                      Z = 1
```

MOVLW - Move literal to W

Syntax: [label] MOVLW k

Description: 8-bit literal **k** is moved to register **W**.

Operation: k -> (W)

Operand: 0 ≤ **k** ≤ 255

Status affected: -

Number of cycles: 1

EXAMPLE 1:

```
....
[label] MOVLW 0x5A
```

```
After instruction: W=0x5A
```

EXAMPLE 2:

```
Const equ 0x40
```

```
[label] MOVLW Const
```

```
Before instruction execution: W=0x10
After instruction:  W=0x40
```

MOVWF - Move W to f

Syntax: [label] MOVWF **f**

Description: The content of register W is moved to register **f**.

Operation: (W) -> f

Operand: 0 ≤ **f** ≤ 127

Status affected: -

Number of cycles: 1

EXAMPLE 1:

```
....
[label] MOVWF OPTION_REG
```

```
Before instruction execution: OPTION_REG=0x20
                              W=0x40
After instruction:  OPTION_REG=0x40
                              W=0x40
```

EXAMPLE 2:

```
....
[label] MOVWF INDF
```

```
Before instruction execution: W=0x17
                              FSR=0xC2, register at address 0xC2 contains
```

```
the value 0x00
After instruction:  W=0x17
                    FSR=0xC2, register at address 0xC2 contains
the value 0x17
```

NOP - No Operation

Syntax: [label] NOP

Description: No operation.

Operation: -

Operand: -

Status affected: -

Number of cycles: 1

EXAMPLE :

```
....
[label] NOP ; 1us delay (oscillator 4MHz)
```

```
Before instruction execution: PC = x
After instruction:  PC = x + 1
```

RETFIE - Return from Interrupt

Syntax: [labels] RETFIE

Description: Return from subroutine. The value is popped from the stack and loaded to the program counter. Interrupts are enabled by setting the bit GIE of the INTCON register.

Operation: TOS -> PC, 1 -> GIE

Operand: -

Status affected: -

Number of cycles: 2

EXAMPLE :

```
....
[label] RETFIE
```

```
Before instruction execution: PC = x
                      GIE (interrupt enable bit of the SATUS
register) = 0
After instruction:  PC = TOS (top of stack)
                      GIE = 1
```

RETLW - Return with literal in W

Syntax: [label] RETLW **k**

Description: 8-bit literal **k** is loaded into register **W**. The value from the top of stack is loaded to the program counter.

Operation: (k) -> W; top of stack (TOP) -> PC

Operand: -

Status affected: -

Number of cycles: 2

EXAMPLE :

```
....
[label] RETLW 0x43
```

```
Before instruction execution: W = x
                      PC = x
                      TOS (top of stack) = x
After instruction:  W = 0x43
                      PC = TOS (top of stack)
                      TOS (top of stack) = TOS - 1
```

RETURN - Return from Subroutine

Syntax: [label] RETURN

Description: Return from subroutine. The value from the top of stack is loaded to the program counter. This is a two-cycle instruction.

Operation: TOS -> program counter PC.

Operand: -

Status affected: -

Number of cycles: 2

EXAMPLE :

```
....
[label] RETURN
```

```
Before instruction execution: PC = x
                      TOS (top of stack) = x
After instruction:  PC = TOS (top of stack)
                      TOS (top of stack) = TOS - 1
```

RLF - Rotate Left f through Carry

Syntax: [label] RLF **f**, **d**

Description: The content of register f is rotated one bit to the left through the Carry flag. If **d** = **w** or **d** = 0, the result is stored in register **W**. If **d** = **f** or **d** = 1, the result is stored in register **f**.

Operation: (f(n)) -> d(n+1), f(7) -> C, C -> d(0);

Operand: 0 ≤ **f** ≤ 127, d[0,1]

Status affected: C

Number of cycles: 1

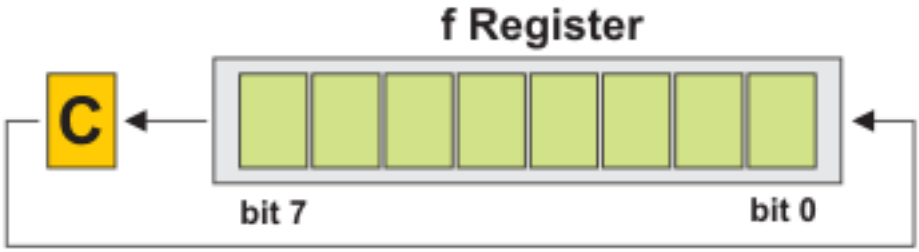


Fig. 9-1 f Register

EXAMPLE 1:

```
....
[label] RLF REG,w
```

```
Before instruction execution: REG = 1110 0110
                              C = 0
After instruction:  REG = 1110 0110
                    W = 1100 1100
                    C = 1
```

EXAMPLE 2:

```
....
[label] RLF REG,f
```

```
Before instruction execution: REG = 1110 0110
                              C = 0
After instruction:  REG = 1100 1100
                    C = 1
```

RRF - Rotate Right f through Carry

Syntax: [label] RRF **f**, **d**

Description: The content of register **f** is rotated one bit right through the Carry flag. If **d** = **w** or **d** = 0, the result is stored in register **W**. If **d** = **f** or **d** = 1, the result is stored in register **f**.

Operation: (f(n)) -> d(n-1), f(0) -> C, C -> d(7);

Operand: 0 ≤ **f** ≤ 127, d -> [0,1]

Status affected: C

Number of cycles: 1

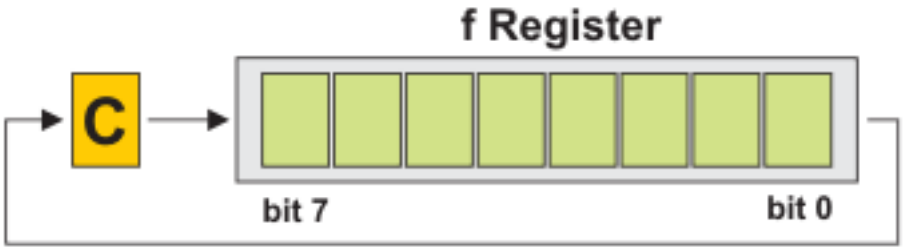


Fig. 9-2 f Register

EXAMPLE 1:

```
....
[label] RRF REG,w
```

```
Before instruction execution: REG = 1110 0110
                      W = x
                      C = 0
After instruction:  REG = 1110 0110
                      W = 0111 0011
                      C = 0
```

EXAMPLE 2:

```
....
[label] RRF REG,f
```

```
Before instruction execution: REG = 1110 0110, C = 0
After instruction:  REG = 0111 0011, C = 0
```

SLEEP - Enter Sleep mode

Syntax: [label] SLEEP

Description: The processor enters sleep mode. The oscillator is stopped. PD bit (**Power Down**) of the STATUS register is cleared. TO bit of the same register is set. The WDT and its prescaler are cleared.

Operation: 0 -> WDT, 0 -> WDT prescaler, 1 -> TO, 0 -> PD

Operand: -

Status affected: TO, PD

Number of cycles: 1

EXAMPLE :

```
....
[label] SLEEP
```

```
Before instruction execution: WDT counter = x
                      WDT prescaler = x
After instruction:    WDT counter = 0x00
                      WDT prescaler = 0
                      TO = 1
                      PD = 0
```

SUBLW - Subtract W from literal

Syntax: [label] SUBLW **k**

Description: The content of register **W** is subtracted from the literal **k**. The result is stored in register **W**.

Operation: k - (W) -> W

Operand: 0 ≤ **k** ≤ 255

Status affected: C, DC, Z

Number of cycles: 1

EXAMPLE :

```
....
[label] SUBLW 0x03
```

```
Before instruction execution: W = 0x01, C = x, Z = x
After instruction:  W = 0x02, C = 1, Z = 0 result is positive

Before instruction execution: W = 0x03, C = x, Z = x
After instruction:  W = 0x00, C = 1, Z = 1 result is 0

Before instruction execution: W = 0x04, C = x, Z = x
After instruction:  W = 0xFF, C = 0, Z = 0 result is negative
```

SUBWF - Subtract W from f

Syntax: [label] SUBWF **f**, **d**

Description: The content of register **W** is subtracted from register **f**.
If **d** = **w** or **d** = **0**, the result is stored in register **W**. If **d** = **f** or **d** = **1**, the result is stored in register **f**.

Operation: (f) - (W) -> d

Operand: 0 ≤ **f** ≤ 127, d [0,1]

Status affected: C, DC, Z

Number of cycles: 1

EXAMPLE :

```
....
[label] SUBWF REG,f
```

```
Before instruction execution: REG = 3, W = 2, C = x, Z = x
After instruction:  REG = 1, W = 2, C = 1, Z = 0 result is
positive

Before instruction execution: REG = 2, W = 2, C = x, Z = x
```

```
After instruction:  REG = 0, W = 2, C = 1, Z = 1 result is 0

Before instruction execution: REG = 1, W = 2, C = x, Z = x
After instruction:  REG = 0xFF, W = 2, C = 0, Z = 0 result is
negative
```

SWAPF - Swap Nibbles in f

Syntax: [label] SWAPF **f**, **d**

Description: The upper and lower nibbles of register **f** are swapped. If **d** = **w** or **d** = **0**, the result is stored in register **W**. If **d** = **f** or **d** = **1**, the result is stored in register **f**.

Operation: f(0:3) -> d(4:7), f(4:7) -> d(0:3);

Operand: 0 ≤ **f** ≤ 127, d [0,1]

Status affected: -

Number of cycles: 1

EXAMPLE 1:

```
....
[label] SWAPF REG,w
```

```
Before instruction execution: REG=0xF3
After instruction:  REG=0xF3
                    W = 0x3F
```

EXAMPLE 2:

```
....
[label] SWAPF REG,f
```

```
Before instruction execution: REG=0xF3
After instruction:  REG=0x3F
```

XORLW - Exclusive OR literal with W

Syntax: [label] **XORLW k**

Description: The content of register **W** is XOR'ed with the 8-bit literal **k** . The result is stored in register **W**.

Operation: (**W**) .XOR. **k** -> **W**

Operand: 0 ≤ **k** ≤ 255

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] XORLW 0xAF
```

```
Before instruction execution: W = 0xB5 ; 1011 0101 (0xB5)
                              ; 1010 1111 (0xAF)
                              ------------------
After instruction:  W = 0x1A ; 0001 1010 (0x1A)
                    Z = 0
```

EXAMPLE 2:

```
Const equ 0x37
[label] XORLW Const
```

```
Before instruction execution: W=0xAF       ; 1010 1111 (0xAF)
                    Const = 0x37 ; 0011 0111 (0x37)
                    -------------------------------
After instruction:  W = 0x98     ; 1001 1000 (0x98)
                    Z = 0
```

XORWF - Exclusive OR W with f

Syntax: [label] XORWF **f**, **d**

Description: The content of register **f** is XOR'ed with the content of register **W**. A bit of result is set only if the corresponding bits of operands are different. If **d** = **w** or **d** = 0, the result is stored in register **W**. If **d** = **f** or **d** = **1**, the result is stored in register **f**.

Operation: (**W**) .XOR. **k** -> **d**

Operand: 0 ≤ **f** ≤ 127, d[0,1]

Status affected: Z

Number of cycles: 1

EXAMPLE 1:

```
....
[label] XORWF REG,f
```

```
Before instruction execution: REG = 0xAF, W = 0xB5 ; 1010 1111
(0xAF)
                                          ; 1011 0101 (0xB5)
                                          ------------------
After instruction:  REG = 0x1A, W = 0xB5 ; 0001 1010 (0x1A)
```

EXAMPLE 2:

```
....
[label] XORWF REG,w
```

```
Before instruction execution: REG = 0xAF, W = 0xB5 ; 1010 1111
(0xAF)
                                          ; 1011 0101 (0xB5)
                                          ------------------
After instruction:  REG = 0xAF, W = 0x1A ; 0001 1010 (0x1A)
```

In addition to the preceding instructions, *Microchip* has also introduced some other instructions. To be more precise, they are not instructions as such, but macros supported by MPLAB. *Microchip* calls them "Special Instructions" since all of them are in fact obtained by

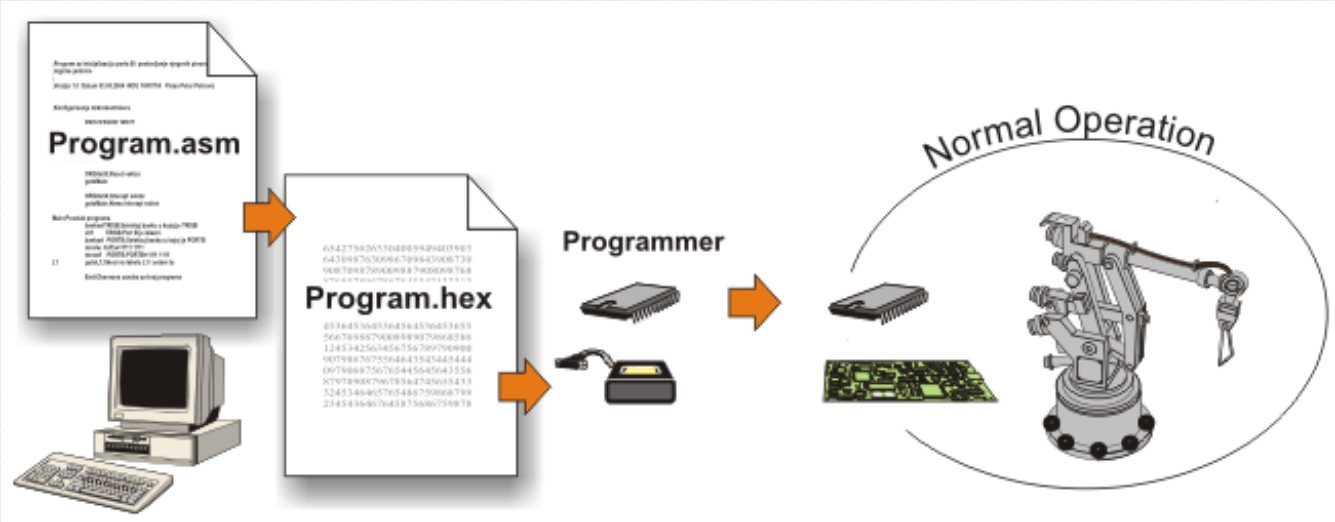
combining already existing instructions.

Instruction		Description	Equivalent Instruction	Status Affected
ADDCF	f,d	Add with carry	BTFSC INCF	STATUS,C
ADDDCF	f,d	Add with Digit Carry	BTFSC INCF	STATUS,DC
B	k	Branch	GOTO	
BC	k	Branch on Carry	BTFSC GOTO	STATUS,C
BDC	k	Branch on Digit Carry	BTFSC GOTO	STATUS,DC
BNC	k	Branch on No Carry	BTFSS GOTO	STATUS,C
BNDC	k	Branch on No Digit Carry	BTFSS GOTO	STATUS,DC
BNZ	k	Branch on No Zero	BTFSS GOTO	STATUS,Z
BZ	k	Branch on Zero	BTFSC GOTO	STATUS,Z
CLRC		Clear Carry	BCF	STATUS,C
CLRDC		Clear Digit Carry	BCF	STATUS,DC
CLRZ		Clear Zero	BCF	STATUS,Z
MOVFW	f	Move File to W	MOVF	
SETC	f	Set Carry	BSF	STATUS,C
SETDC		Set Digit Carry	BSF	STATUS,DC
SETZ		Set Zero	BSF	STATUS,Z
SKPC		Skip on Carry	BTFSS	STATUS,C
SKPDC		Skip on Digit Carry	BTFSS	STATUS,DC
SKPNC		Skip on No Carry	BTFSC	STATUS,Z
SKPNDC		Skip on No Digit Carry	BTFSC	STATUS,DC
SKPNZ		Skip on Non Zero	BTFSC	STATUS,Z
SKPZ		Skip on Zero	BTFSS	STATUS,Z
SUBCF	f, d	Subtract Carry from File	BTFSC DECF	STATUS,C
SUBDCF	f, d	Subtract Digit Carry from File	BTFSC DECF	STATUS,DC
TSTF	f	Test File	MOVF	

Previous Chapter | Table of Contents | Next Chapter

Appendix A: Programming a Microcontroller

Microcontrollers and humans communicate through the medium of the programming language called Assembly language. The word Assembler itself does not have any deeper meaning, it corresponds to the names of other languages such as English or French. More precisely, assembly language is only a passing solution. In order that the microcontroller can understand a program written in assembly language, it must be compiled into a language of zeros and ones. Assembly language and Assembler do not have the same meaning. The first one refers to the set of rules used for writing program for the microcontroller, while the later refers to a program on a personal computer used to translate assembly language statements into the language of zeros and ones. A compiled program is also called Machine Code. A "Program" is a data file stored on a computer hard disc (or in memory of the microcontroller, if loaded) and written according to the rules of assembly or some other programming language. Assembly language is understandable for humans because it consists of meaningful words and symbols of the alphabet. Let us take, for example the command "RETURN" which is, as its name indicates, used to return the microcontroller from a subroutine. In machine code, the same command is represented by a 14-bit array of zeros and ones understandable by the microcontroller. All assembly language commands are similarly compiled into the corresponding array of zeros and ones. A data file used for storing compiled program is called an "executive file", i.e. "HEX data file". The name comes from the hexadecimal presentation of a data file and has a suffix of "hex" as well, for example "probe.hex". After has been generated, the data file is loaded into the microcontroller using a programmer. Assembly language programs may be written in any program for text processing (editor) able to create ASCII data files on a hard disc or in a specialized work environment such as MPLAB described later.



ELEMENTS OF ASSEMBLY LANGUAGE

A program written in assembly language consists of several elements being differently interpreted while compiling the program into an executable data file. The use of these elements requires strict rules and it is necessary to pay special attention to them during program writing in order to avoid errors.

ASSEMBLY LANGUAGE SYNTAX

As mentioned, it is necessary to observe some specific rules in order to enable the process of compiling into executive HEX code to run without errors. Compulsory rules explaining how sequences of expressions are put together to form the statements that make up an assembly language program are called syntax. There are only several of them:

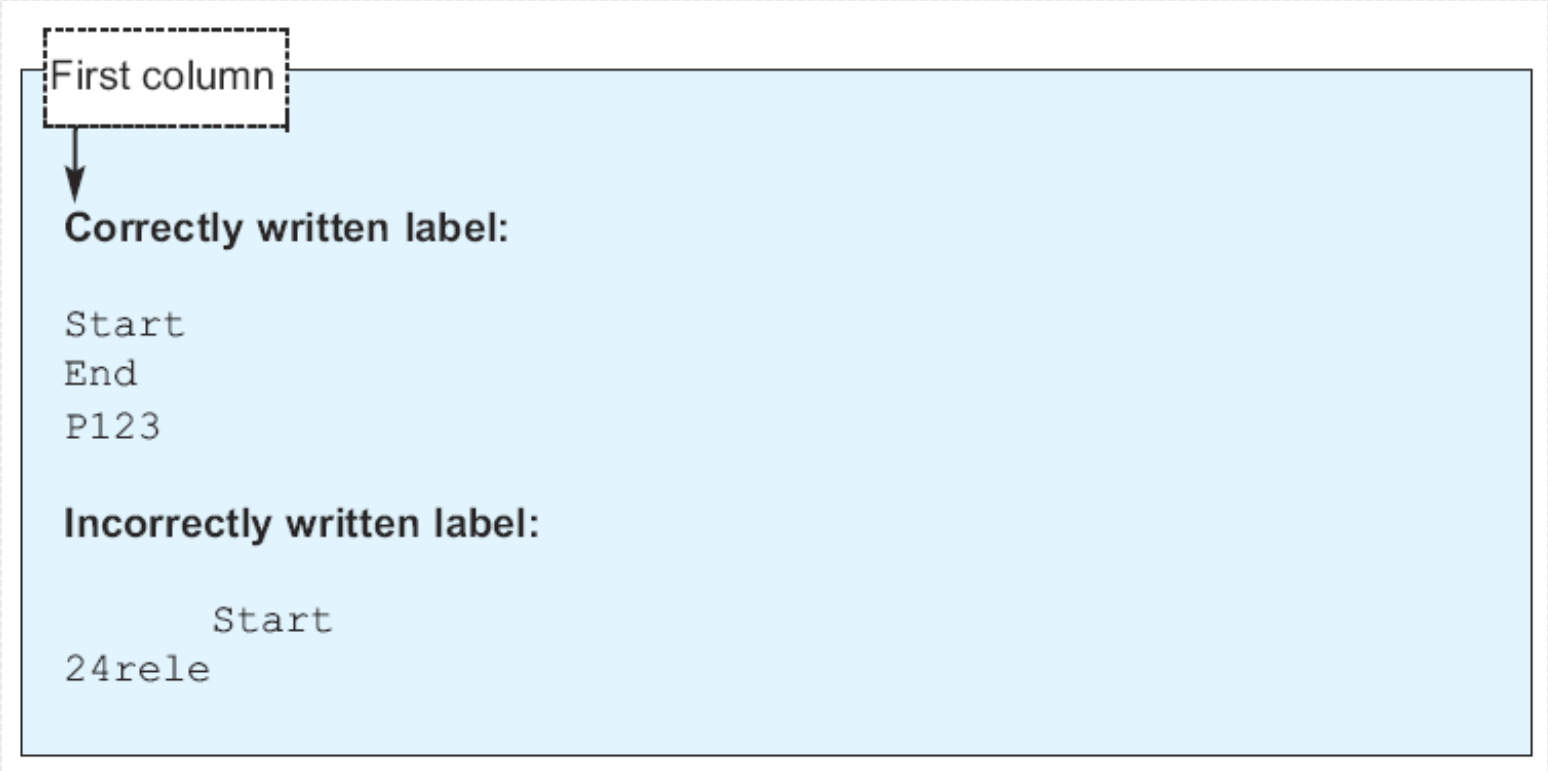
- Every program line may consist of a maximum of 255 characters;
- Every program line that is to be compiled must start with a symbol, label, mnemonics or directive;
- Text following the mark ";" in a program line represents a comment which is ignored by the assembler (not compiled); and
- All the elements of one program line (labels, instructions etc.) must be separated by at least one space character. For the sake of better clearness, a push-button TAB is commonly used instead of it, so that it is easy to delimit columns with labels, directives etc. in a program.

LABELS

A label represents a textual version of some address in ROM or RAM memory. Each label has to start in the first column with a letter of alphabet or "_" and may consist of maximum of 32 characters. Besides, it is easily used:

- It is sufficient to enter the name of a label instead of a 16-bit address in instruction which calls some subroutine or a jump. The label with the same name should also be written at the beginning of a program line in which a subroutine starts or where a jump should be executed. As a general rule, labels have easily recognizable names.

During program compiling, the assembler will automatically replace the labels by the corresponding addresses.



COMMENTS

Acomment is often an explanatory text written by the programmer in order to make a program clearer and easier to understand. It is not necessary to comment every line. When three or four lines of code work together to accomplish some higher level task, it is better to have a single higher level comment for the group of lines. Therefore, it is added if needed and has to start with ";". Comments added to assembly source code are not compiled into machine code.

INSTRUCTIONS

Instructions are defined for each microcontroller family by the manufacturer. Therefore, it is up to the user to follow the rules of their usage. The way of writing instructions is also called instruction syntax. The instructions "`movlp`" and "`gotto`", in the following example, are recognized by the PIC16F887 microcontroller as an error since they are not correctly written.

Correctly written commands:

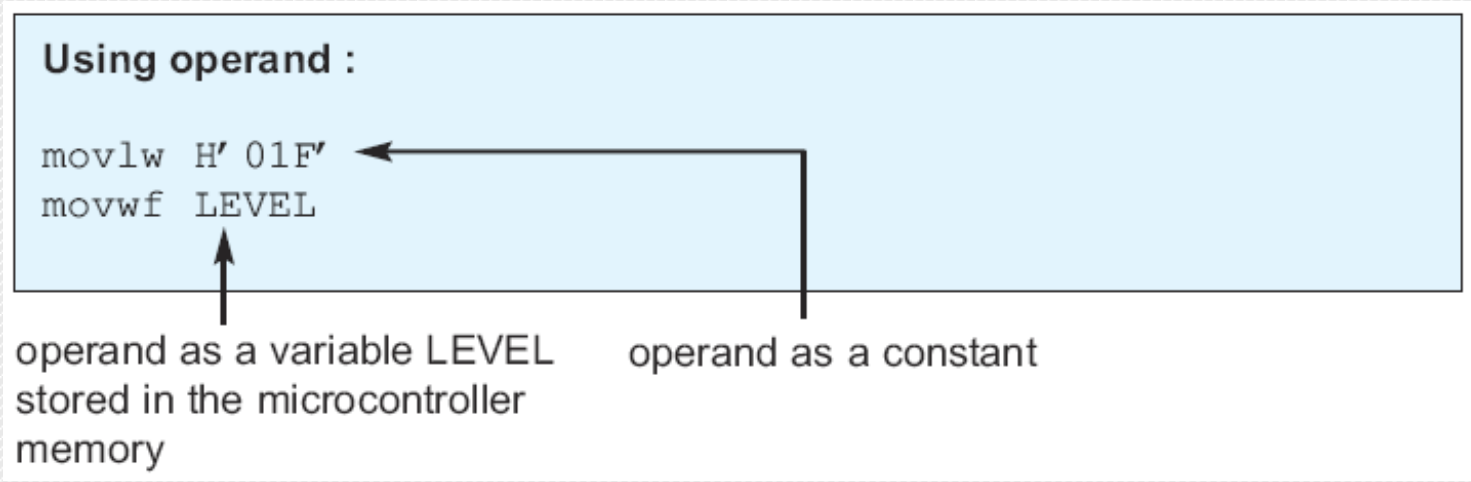
```
movlw H' FF'
goto Start
```

Incorrectly written commands:

```
movlp H' FF'
gotto Start
```

OPERANDS

An operand is a value (an argument) upon which the instruction, named by mnemonic, operates. The operands may be a register, a variable, a literal constant, a label or a memory address.



DIRECTIVES

Unlike instructions being written to on-chip program memory after compilation, directives are commands of assembly language itself and do not directly affect the operation of the microcontroller. Some of them must be used in every program while others are only used to facilitate or enhance the operation. Directives are written to the column reserved for instructions. The rule which must be observed allows only one directive per program line.

This section covers only a few of the most commonly used directives. It would certainly take up too much space and time to describe all the directives recognized by the MPLAB program. Anyway, a complete list containing all directives which the MPLAB assembler can understand is provided in *Help*.

PROCESSOR Directive

This directive must be written at the beginning of each program. It defines the type of the microcontroller which the program is written for. For example:

```
Processor 16f887
```

EQU directive

This directive is used to replace a numeric value by a symbol. In this way, some a specific location in memory is assigned a name. For example:

```
MAXIMUM EQU H’25’
```

This means that a memory location at address 25 (hex.) is assigned the name "MAXIMUM". Every appearance of the label "MAXIMUM" in the program will be interpreted by the assembler as the address 25 (MAXIMUM = H’25’). Symbols may be defined this way only once in a program. That this directive is mostly used at the beginning of the program.

ORG directive

This directive specifies a location in program memory where the program following directive is to be placed. For example:

```
          ORG      0x100
START     ...      ...
          ...
          ORG      0x1000
TABLE     ...
          ...
```

This program starts at location 0x100. The table containing data is to be stored at location 1024 (1000h).

END directive

Each program must be ended by using this directive. Once a program encounters this directive, the assembler immediately stops compiling. For example:

```
...
END ;End of program
```

$INCLUDE directive

The name of this directive fully indicates its purpose. During compiling, it enables the assembler to use data contained in another file on a computer hard disc. For example:

```
...
#include <p16f887.inc>
```

CBLOCK and ENDC directives

All variables (their names and addresses) that will be used in a program must be defined at the beginning of the program. Because of this it is not necessary to specify the address of each specified variable later in the program. Instead, it is enough to specify the address of the first one by using directive CBLOCK and list all others afterwards. The compiler automatically assigns these variables the corresponding addresses as per the order they are listed. Lastly, the directive ENDC indicates the end of the list of variables.

```
CBLOCK    0x20
          START    ; address 0x20
          RELE     ; address 0x21
          STOP     ; address 0x22
          LEFT     ; address 0x23
```

```
          RIGHT    ; address 0x24
ENDC
          ...
```

IF, ENDIF and ELSE directives

These directives are used to create so called conditional blocks in a program. Each of these blocks starts with the directive IF and ends with the directive ENDIF or ELSE. A statement or a symbol (in parentheses) following the directive IF represents a condition which determines which part of the program is to be compiled:

- If the statement is correct or the value of a symbol is equal to one, program compiles all instructions written before directive ELSE or ENDIF; and
- If the statement is not correct or the value of a symbol is equal to zero, only instructions written after directives ELSE or ENDIF are to be compiled.

Example 1:

```
IF    (VERSION>3)
       CALL    Table_2
       CALL
ENDIF
       ...
```

If the program is released after the version 3 (statement is right) then subroutines "Table 2" and "Extension" are executed. If the statement in parentheses is wrong (VERSION<3), two instructions calling subroutines are ignored and will not be compiled therefore.

Example 2:

If the value of symbol "Model" is equal to one then first two instructions after directive IF are compiled as well as instructions after directive ENDIF (all instructions between ELSE and ENDIF are ignored). Otherwise, if Model=0 then instructions between IF and ELSE are ignored, whereas instructions after directive ELSE are compiled.

```
IF    (Model)
      MOVFW      BUFFER
      MOVWF      MAXIMUM
ELSE
      MOVFW      BUFFER1
      MOVWF      MAXIMUM
ENDIF
      ...
```

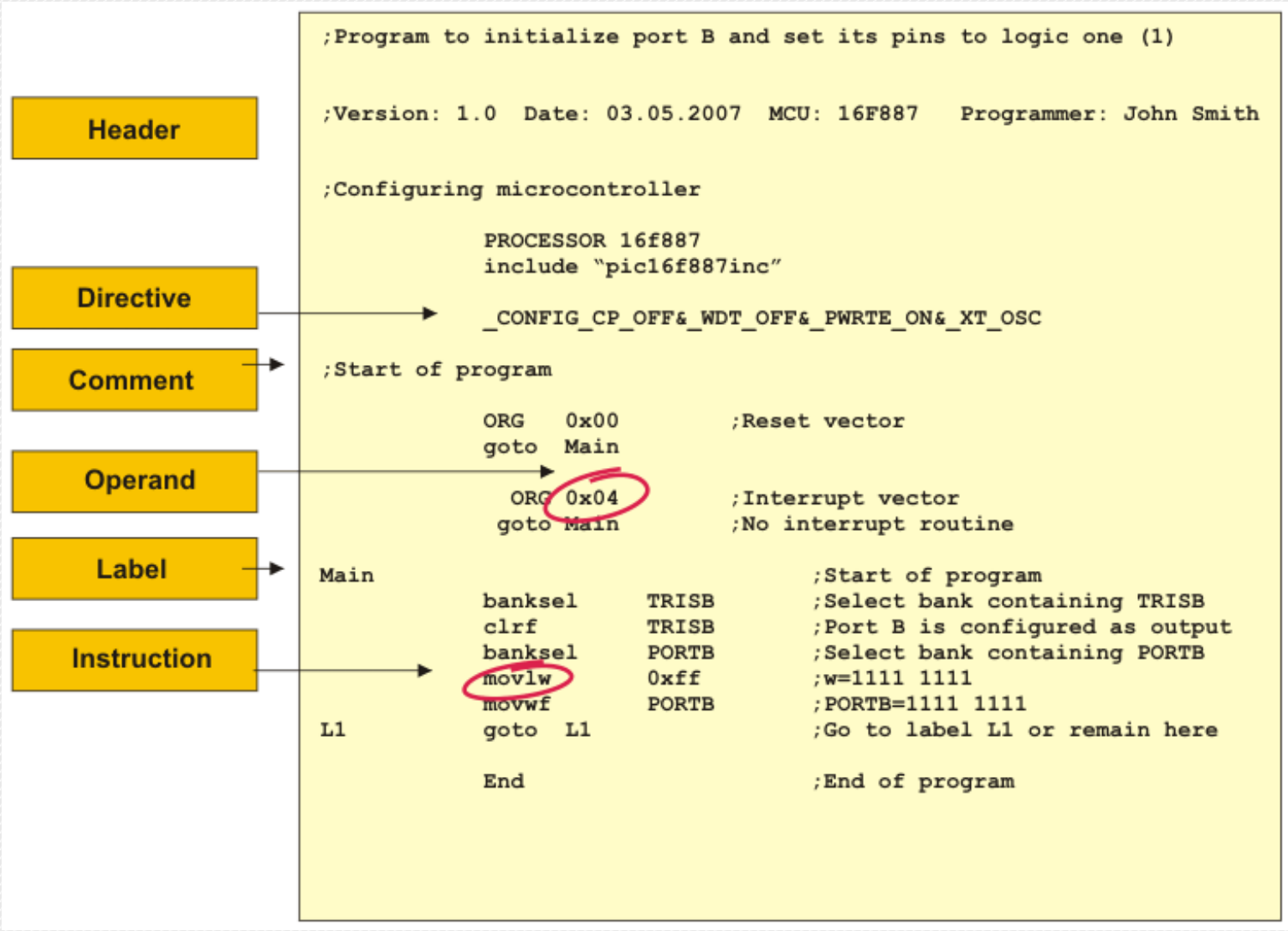
BANKSEL directive

In order to access an SFR register it is necessary to select the appropriate bank in RAM memory by using bits RP0 and RP1 of the STATUS register. This directive is used in this case. Simply, since "inc" data file contains the list of all registers along with their addresses, the assembler knows which bank corresponds to which register. After encountering this directive, assembler selects the bits RP0 and RP1 for the specified register on its own. For example:

```
          ...
BANKSEL   TRISB
          CLRF TRISB
          MOVLW B'01001101'
BANKSEL   PORTB
          MOVWF PORTB
          ...
```

EXAMPLE OF HOW TO WRITE A PROGRAM

The following example illustrates what a simple program written in assembly language looks like.




```
;Program to initialize port B and set its pins to logic one (1)

;Version: 1.0  Date: 03.05.2007  MCU: 16F887   Programmer: John Smith

;Configuring microcontroller

            PROCESSOR 16f887
            include "pic16f887inc"

            _CONFIG_CP_OFF&_WDT_OFF&_PWRTE_ON&_XT_OSC

;Start of program

            ORG   0x00          ;Reset vector
            goto  Main

              ORG 0x04          ;Interrupt vector
             goto Main          ;No interrupt routine

Main                                    ;Start of program
            banksel     TRISB           ;Select bank containing TRISB
            clrf        TRISB           ;Port B is configured as output
            banksel     PORTB           ;Select bank containing PORTB
            movlw       0xff            ;w=1111 1111
            movwf       PORTB           ;PORTB=1111 1111
L1          goto  L1                    ;Go to label L1 or remain here

            End                         ;End of program
```

Apart from the regular rules of assembly language, there are also some unwritten rules which should be observed during program writing. One of them is to write in a few words at the beginning of a program what the program's name is, what it is used for, version, release date, type of the microcontroller it is written for and the name of the programmer. Since this information is not of importance for the assembler, it is written as a comment which always starts with semicolon ';' and can be written in a new line or immediately after a command.

After writing this general comment, it is time to select the microcontroller by using directive PROCESSOR. This directive is followed by another one used to include all the definitions of the PIC16F887 microcontroller's internal registers in the program. These definitions are nothing but the ability to address port B and other registers as PORTB instead of 06h, which makes the program clearer and more legible.

In order that the microcontroller will operate properly, a several parameters such as the type of oscillator, state of the watch-dog and internal reset circuit must be defined. It is done by utilizing the following directive:

```
_CONFIG _CP_OFF&_WDT_OFF&PWRTE_ON&XT_OSC
```

When all necessary elements are defined, the process of program writing can start. First and foremost, it is necessary to specify the address from which the microcontroller starts when the power goes on (`org 0x00`) as well as the address from which the program proceeds with execution if an interrupt occurs (`org 0x04`). Since this program is very simple, it is

enough to use command "`goto Main`" in order to direct the microcontroller to the beginning of the program. The next command selects memory bank 1 in order to enable access to the TRISB register to configure port B as output (`banksel TRISB`). The main program ends by selecting memory bank 0 and setting all port B pins to logic one (1)(`movlw 0xFF`, `movwf PORTB`).

It is necessary to create a loop to keep program from "getting lost" in case an error occurs. For this purpose, there is an endless loop executed all the time while the microcontroller is switched on.

"`end`" is required at the end of every program to inform the assembler that there are no more commands to be compiled.

DATA FILES RESULTING FROM PROGRAM COMPILING

The result of compiling a program written in assembly language are data files. The most important and most commonly used data files are:

- Executive data file (Program_Name.HEX);
- Error data file (Program_Name.ERR); and
- List data file (Program_Name.LST).

The first file contains compiled program which is loaded into the microcontroller. Its contents give no information of importance to the programmer so it will not be discussed here. The second file contains errors made in writing process and detected by the compiler during compiling process. Errors can be detected in list data file, which takes more time, so the error data file is more suitable for long programs.

The third file is the most useful for the programmer. It contains lots of information on commands and variables locations in on-chip memory as well as error signalization. There is a symbol table at the end of each data file list containing all the symbols used in a program. Other useful elements of list data file are memory usage maps and error statistics provided at the very end of the file list.

MACROS AND SUBROUTINES

The same sequence of computing instructions is usually used repeatedly within a program. Assembly language is very demanding. The programmer is required to take care of the last little detail when writing a program, because only one wrong command or label name may cause the program to not work properly or it may not work at all. Therefore, it is less tedious and less error-prone to use a sequence of instructions as a single program statement which works properly for sure. To implement this idea, macros and subroutines are used.

MACROS

A macro contains programmer-defined symbols that stand for a sequence of text lines. It is defined by using directive **macro** which names macro and arguments if needed. Macro must be defined prior it is used. Once a macro has been defined, its name may be used in the program.When the assembler encounters macro's name, it replaces it by the appropriate sequence of instructions and processes them just as though they have appeared in the program. Many different macro-instructions are available for various purposes, eliminating some of the repetitiveness of the programming, as well as simplifying the writing, reading and understanding of the program. The simplest use of macros may be giving a name to an instruction sequence being repeated. Let us take, for example, global interrupt enable procedure, SFRs' bank selection.

```
macro_name macro arg1, arg2...
           ...
           sequence of instructions
           ...
           endm
```

The following example shows four macros. The first two macros select banks, the third one enables interrupt, whereas the fourth one disables interrupt.

```
bank0 macro               ; Macro bank0
           bcf STATUS, RP0 ; Reset RP0 bit
           bcf STATUS, RP1 ; Reset RP1 bit
           endm            ; End of macro
```

```
bank1 macro                ; Macro bank1
          bsf STATUS, RP0  ; Set RP0 bit
          bcf STATUS, RP1  ; Reset RP1 bit
          endm             ; End of macro
enableint macro            ; Global interrupt enable
          bsf INTCON,7     ; Set bit
          endm             ; End of macro
disableint macro           ; Global interrupt disable
          bcf INTCON,7     ; Reset bit
          endm             ; End of macro
```

Macros defined in this way are saved in a particular data file with extension INC which stands for INCLUDE data file. As seen, these four macros do not have arguments. However, macros may include arguments if needed.

The following example shows macros with arguments. Pin is configured as input if the corresponding bit of the TRIS register is set to logic one (bank1). Otherwise, it is configured as output.

```
input macro arg1,arg2      ;Macro Input
       bank1               ;Bank containing TRIS registers
       bsf arg1,arg2       ;Set the specified bit (1=Input)
       bank0               ;Macro for bank 0 selection
       endm                ;End of macro

output macro arg1,arg2     ;Macro Output
       bank1               ;Bank containing TRIS registers
       bcf arg1,arg2       ;Clear the specified bit (0=Output)
       bank0               ;Macro for bank 0 selection
       endm                ;End of macro
```

Macro with arguments may be called in the following way:

```
...
output TRISB,7 ;Pin RB7 is configured as output
...
```

When calling this macro, the first specified argument TRISB is replaced by the first argument **arg1** in macro definition. Similarly, number 7 is replaced by the argument **arg2**, and the following code is generated:

```
...
bsf STATUS, RP0   ;Set RP0 bit = BANK1
bcf STATUS, RP1   ;Reset RP0 bit = BANK1

bcf TRISB,7       ;Configure RB7 as output

bcf STATUS,RP0    ;Clear RP0 bit = BANK0
bcf STATUS,RP1    ;Clear RP1 bit = BANK0
...
```

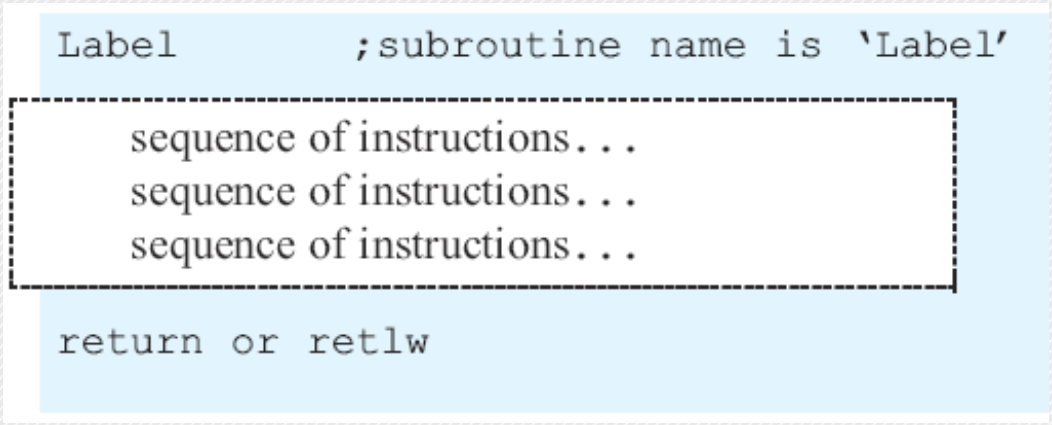
It is clear at first sight that the program becomes more legible and flexible by using macros. The main disadvantage of macro is that it occupies a lot of memory space because every macro name in a program is replaced by its predefined code. Owing to the fact that programs often use macro, everything is more complicated if it is long.

```
callc macro label  ;Macro callc
local Exit         ;Define local Label within macro
      bnc   Exit   ;If C=0 jump to Exit
      call  label  ;If C=1 call subroutine at address Label(out of macro)
Exit               ;Local Label within macro
      endm         ;End of macro
```

In the event that a macro has labels, they must be defined as local ones by using directive local. The given example contains macro which calls a subroutine (`call label` in this case) if the *Carry* bit of the STATUS register is set. Otherwise, the first following instruction is executed.

SUBROUTINES

Asubroutine contains a sequence of instructions, begins with a label (subroutine_name) and ends with command *return* or *retlw*. The main difference comparing to macro is that subroutine is not replaced by its code in the program, but program jumps to subroutine to execute it. It happens every time the assembler encounters command *call Subroutine_name* in the program. On the command *return*, it leaves a subroutine and continues execution from where it left off the main program. Subroutine may be defined both prior to or upon the call.



As seen, concerning macros, the input and output arguments are of great importance. Concerning subroutines, it is not possible to define arguments within the subroutine itself. However, variables predefined in the main program may be used as subroutine arguments.

A logical sequence of events is as follows: defining variables, calling subroutine which uses them and at the end reading variables changed upon the execution of subroutine.

The program in the following example performs addition of two 2-byte variables ARG1 and ARG2 and moves result to the variable RES. When 2-byte variables are used, it is necessary to define higher and lower byte for each of them. The program itself is very simple. It first adds lower bytes of variables ARG1 and ARG2 and higher afterwards. If the sum of addition of two lower bytes is greater than 255 (maximal byte value) the remainder is added to the RESH variable.

```
; Program to add two 16-bit numbers
; Version: 1.0 Date: April 25, 2007 MCU:PIC16F887

      PROCESSOR 16f887 ; Defining processor
      #include "p16f887.inc" ; Microchip INC database
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      Cblock  0x20     ; Beginning of RAM
      ARG1H            ; Argument 1 higher byte
      ARG1L            ; Argument 1 lower byte
      ARG2H            ; Argument 2 higher byte
      ARG2L            ; Argument 2 lower byte
      RESH             ; Result higher byte
      RESL             ; Result lower byte
      endc             ; End of variables
      ORG     0x00     ; Reset vector
      goto    Start
Start                  ; Write values to variables
      movlw   0x01     ; ARG1=0x0104
      movwf   ARG1H
      movlw   0x04
      movwf   ARG1L
      movlw   0x07     ; ARG2=0x0705
      movwf   ARG2H
      movlw   0x05
      movwf   ARG2L
```

```
Main                    ; Main program
      call    Add16     ; Call subroutine Add16
Loop  goto    Loop      ; Remain here
Add16                   ; Subroutine to add two 16-bit numbers
      clrf    RESH      ; RESH=0
      movf    ARG1L,w   ; w=ARG1L
      addwf   ARG2L,w   ; w=w+ARG2L
      movwf   RESL      ; RESL=w
      btfsc   STATUS,C  ; Is the result greater than 255?
      incf    RESH,f    ; If greater, increment RESH by one

      movf    ARG1H,w   ; w=ARG1H
      addwf   ARG2H,w   ; w=w+ARG2
      addwf   RESH,f    ; RESH=w
      return            ; Return from subroutine
      end               ; End of program
```

In Short
The main difference between macros and subroutines is that macro is after compiling replaced by its code (enables the programmer to type less). It may also have arguments while subroutine uses less memory, but does not have arguments.

MPLAB

MPLAB is a Windows program package which enables easy program writing as well as easy program development. It is best to describe it as development environment for a standard program language designed for PC programming. MPLAB technically simplifies some operations consisting of a lot of parameters, which, until the IDE environment* appeared, were executed from the command line. However, tastes are different and there are some programmers who prefer standard editors and command line compilers. Every program written in MPLAB is clear, but there are also help documentation- just in case.

INSTALLING MPLAB

MPLAB consists of several parts:

- The program which sorts data files of the same project into one group (**Project Manager**);
- program for text generating and processing (**Text Editor**); and
- simulator used to simulate the operation of a program loaded into the microcontroller.

Besides, there are also built in programmers such as PICStart Plus and ICD *(In Circuit Debugger)* that can be used to program software into PIC microcontroller device. Since not being the subject of this book, they are mentioned as options only.

In order to start MPLAB, your PC should contain:

- PC compatible computer belonging to class 486 or better;
- Any Windows operating system;
- VGA graphic card;
- 8MB memory (32MB recommended);
- 200MB available hard disc; and
- A mouse.

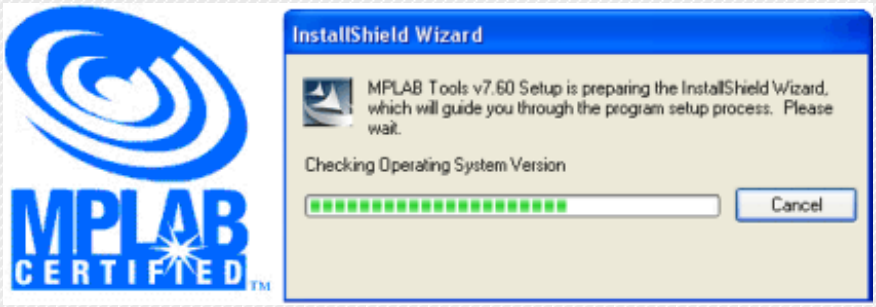
MPLAB installation comes first. Data files from MPLAB CD should be copied to a hard disc. The process of installation is similar to almost all other Windows program installations. First of all a welcome window appears, then options to select and at last installation itself. A message notifying that the program is successfully installed and ready for use appears. Are you ready?

Steps to follow prior the installation:

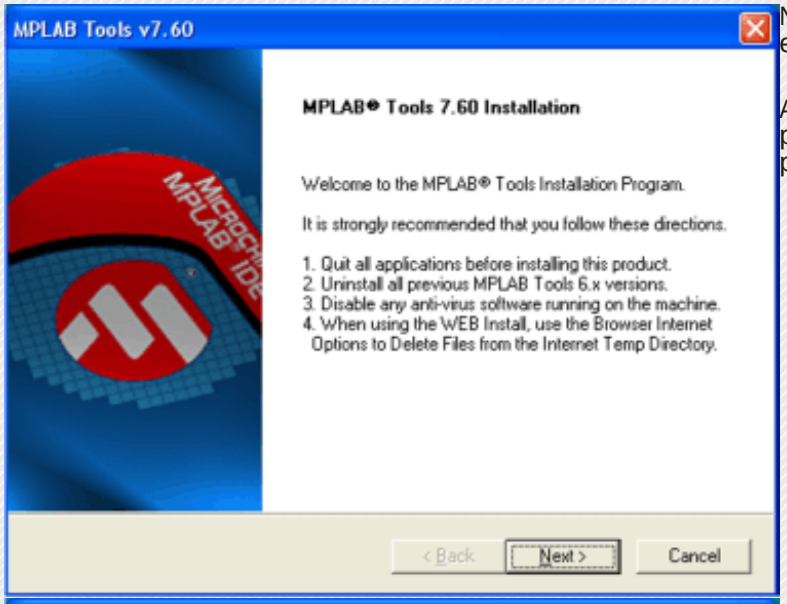
1. Start Microsoft Windows;
2. Insert the CD into CD ROM;
3. Click START and select option RUN;
4. Click BROWSE and select CD ROM drive; and
5. Find folder MPLAB on CD ROM.

Everything is ready now to start installation. The following pictures describe the installation steps.

Click on this icon to start up the process...

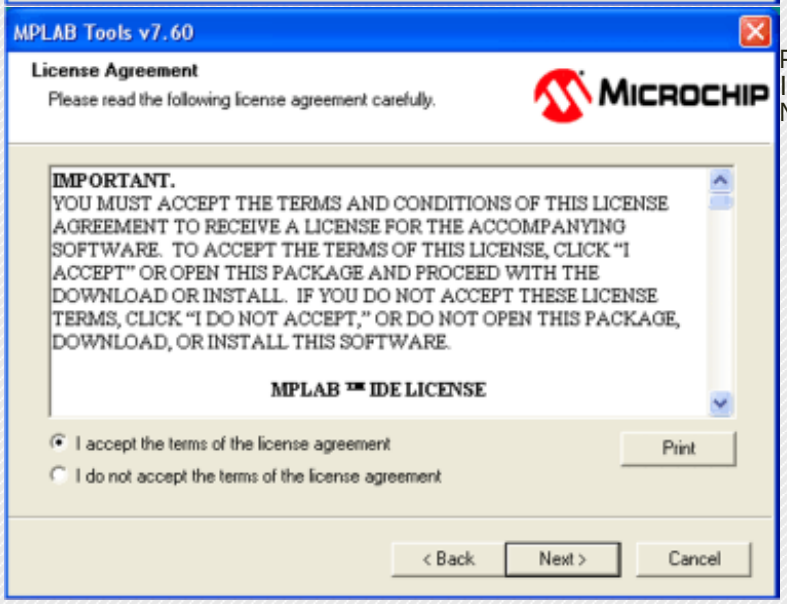


Something is going on... The picture coming up indicates that the process of installation has just started!

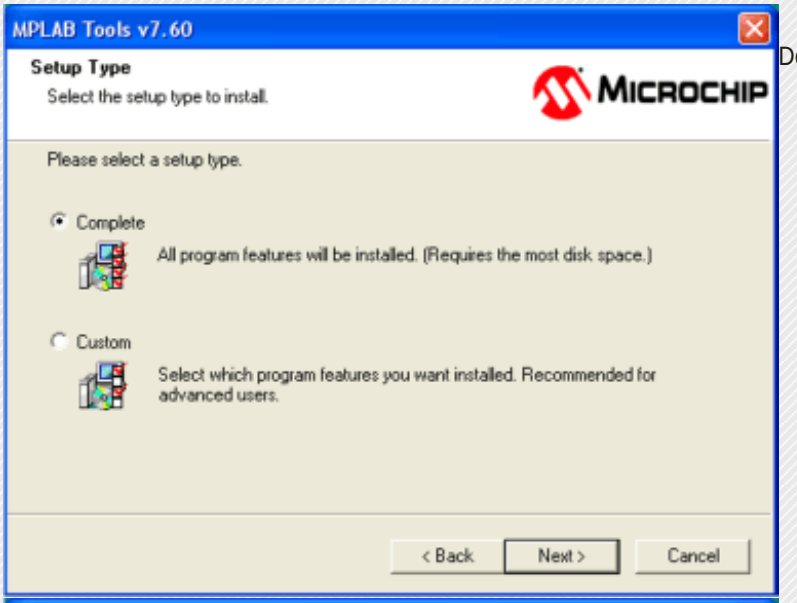


Next window contains the word "Welcome". Need explanation?

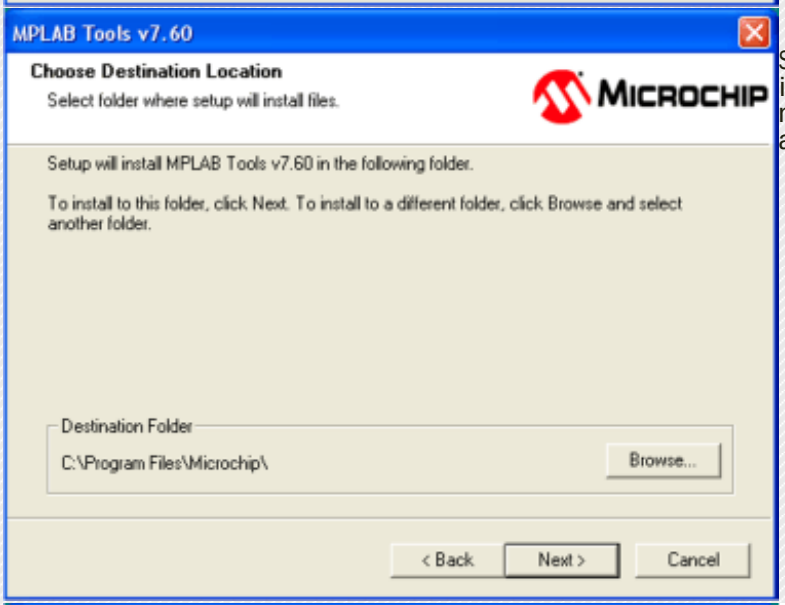
Actually, the program reminds you to close all active programs in order to not interfere with the installation process. Next- of course!



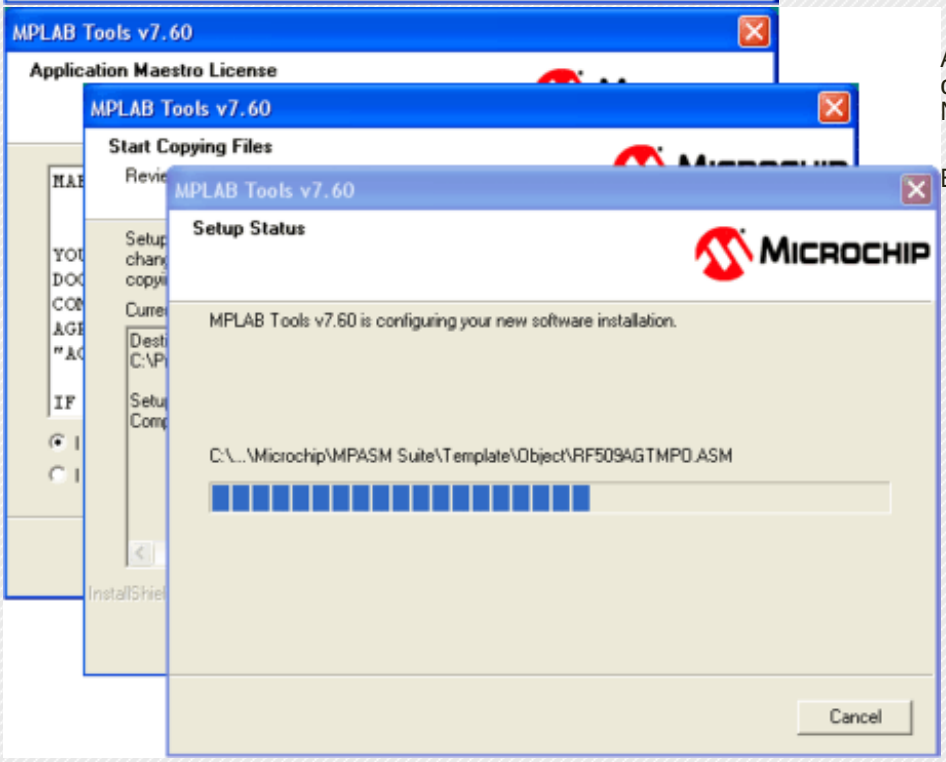
Prior to continue, you have to accept the MPLAB software license conditions. Select the option "I accept" and click NEXT.



Do you want to install the entire software? Yes. Next...

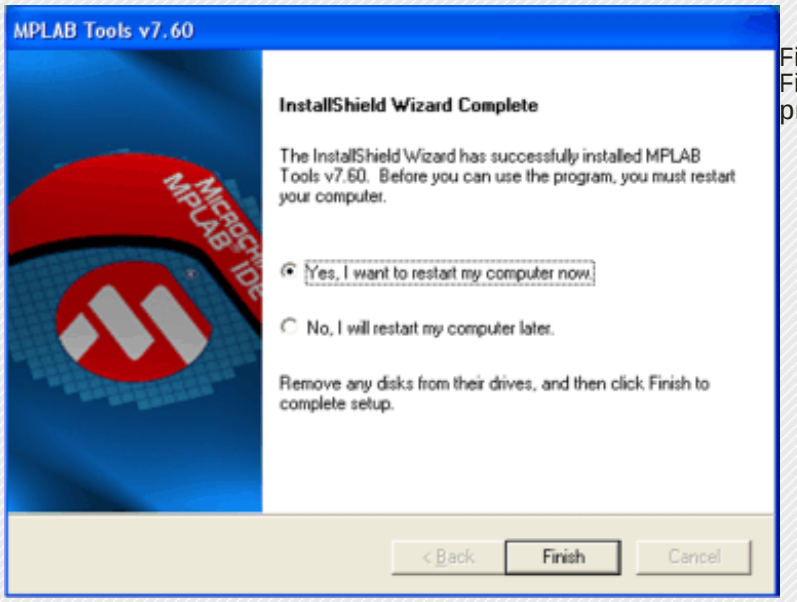


Similar to other programs, MPLAB should be also installed into a folder. It may be any folder on any hard disc. If it is not necessary to make changes, select the specified address and click Next.



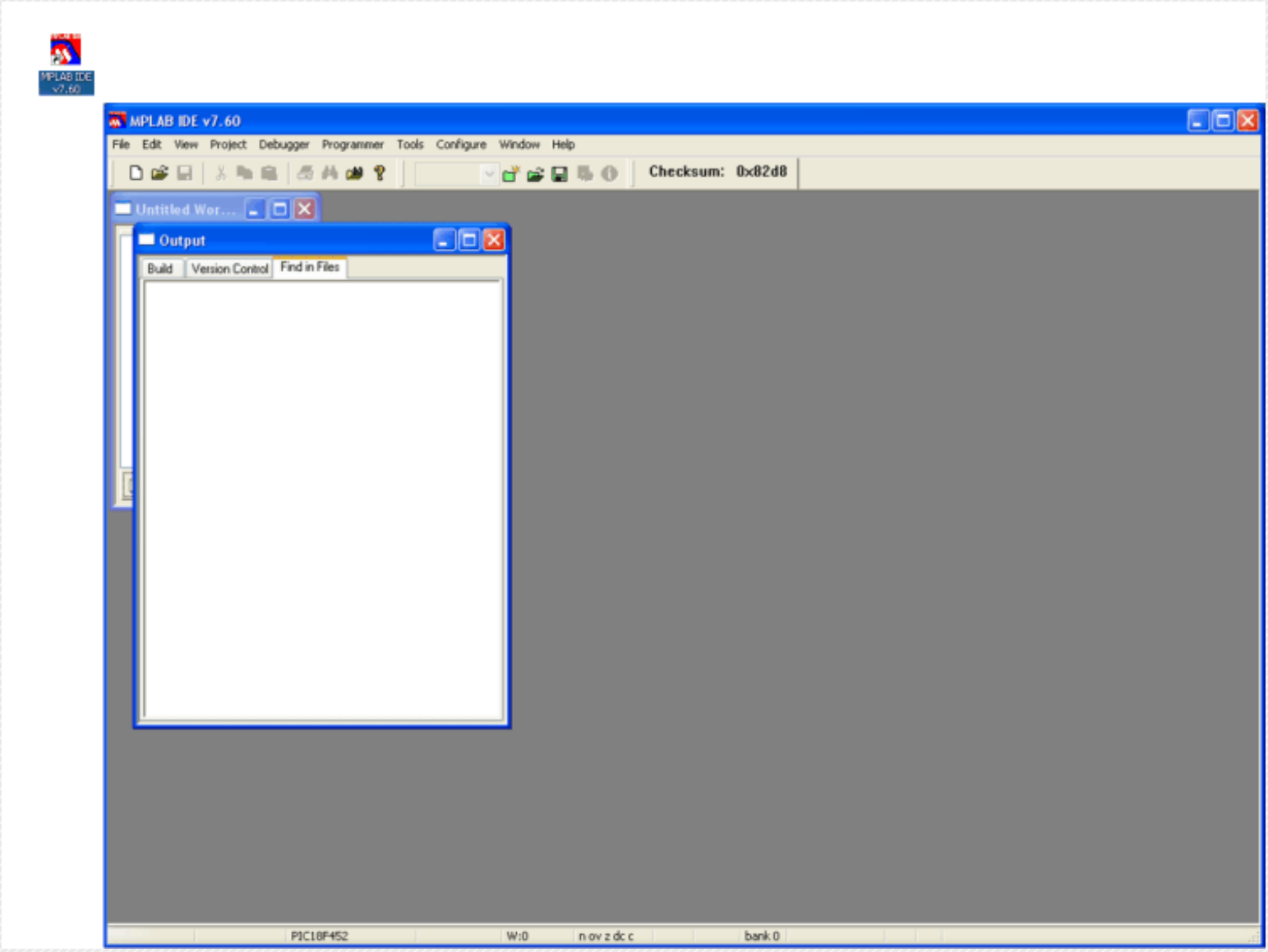
Another license, another acceptance of options specified by the computer... Next, Next...

Be patient!



Finally! This is what you have been waiting for. Click Finish. The computer will be restarted along with the program saved on hard disc. Everything is OK!

Click the MPLAB desktop icon in order to start the program and learn about it.



As seen, MPLAB is similar to most Windows programs. Apart from the working area, there are menus (contains options: File, Edit etc.), toolbars (contains different icons) and a status bar at the bottom of the window. Similar to Windows,

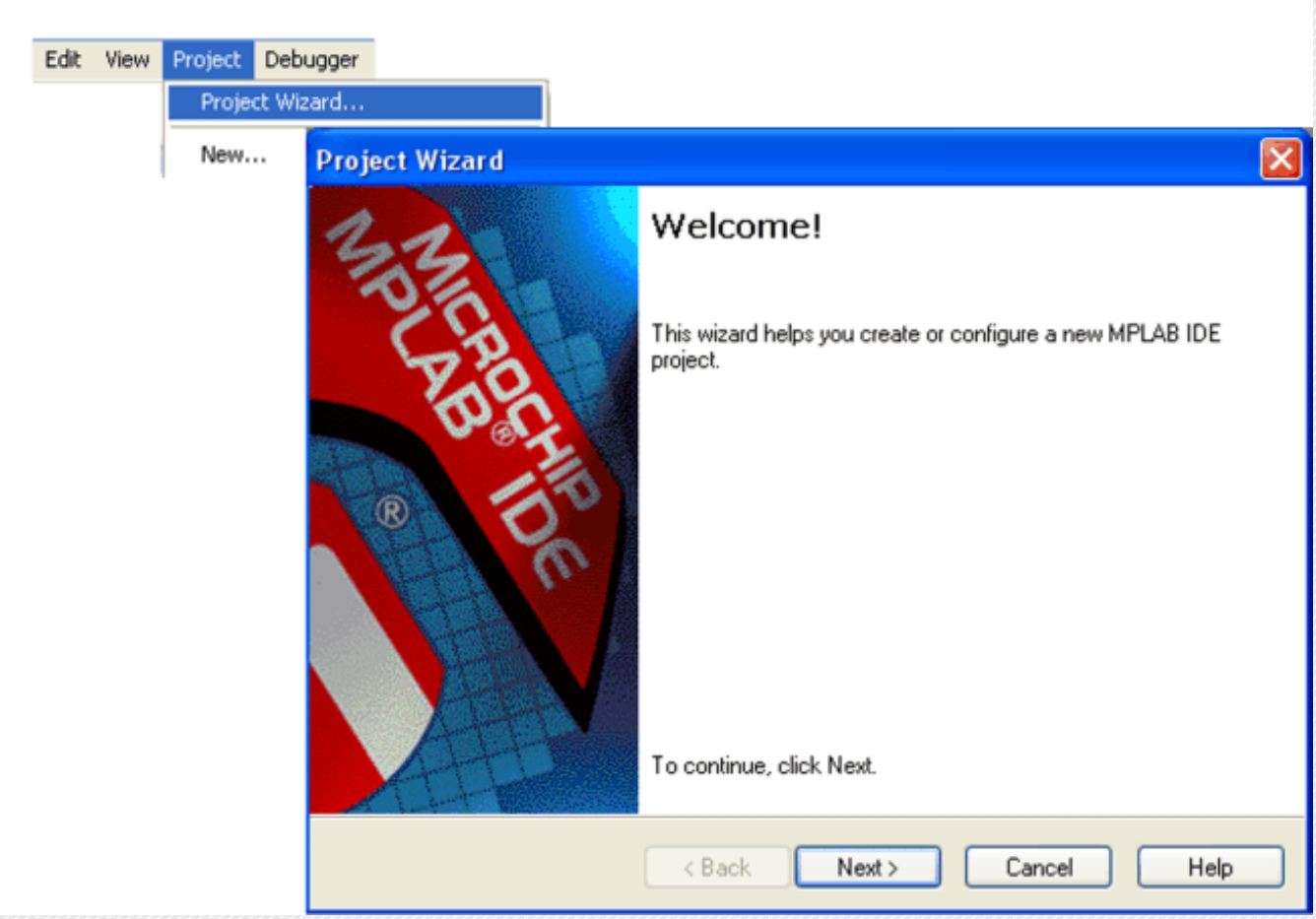
there is a rule to have shortcuts for the most commonly used program options created in order to easily access them and speed up operation therefore. These shortcuts are actually icons below the menu bar. In other words, all options contained in the toolbar are contained in the menu too.

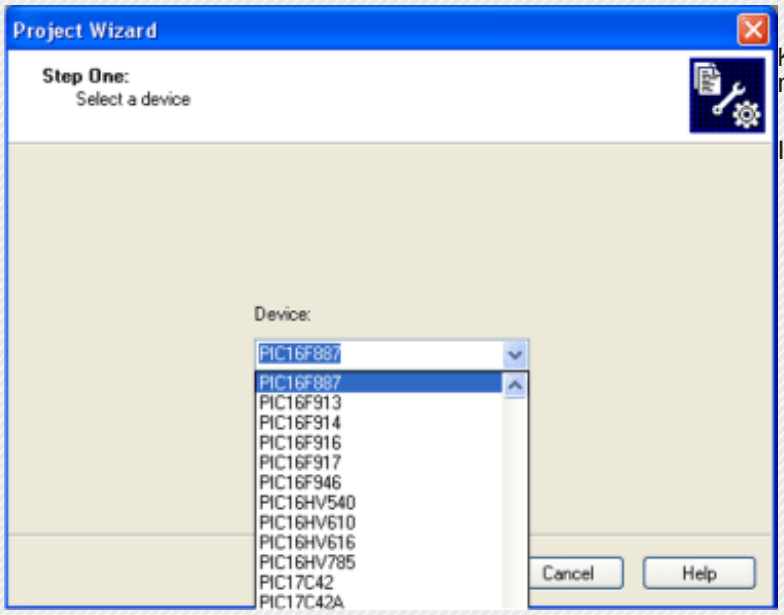
PROJECT-MAKING

Follow these steps to prepare program for loading into the microcontroller:

1. Make a project;
2. Write a program; and
3. Compile it.

In order to make a project, it is necessary to click the option "PROJECT" and then "PROJECT WIZARD". A welcome window appears.

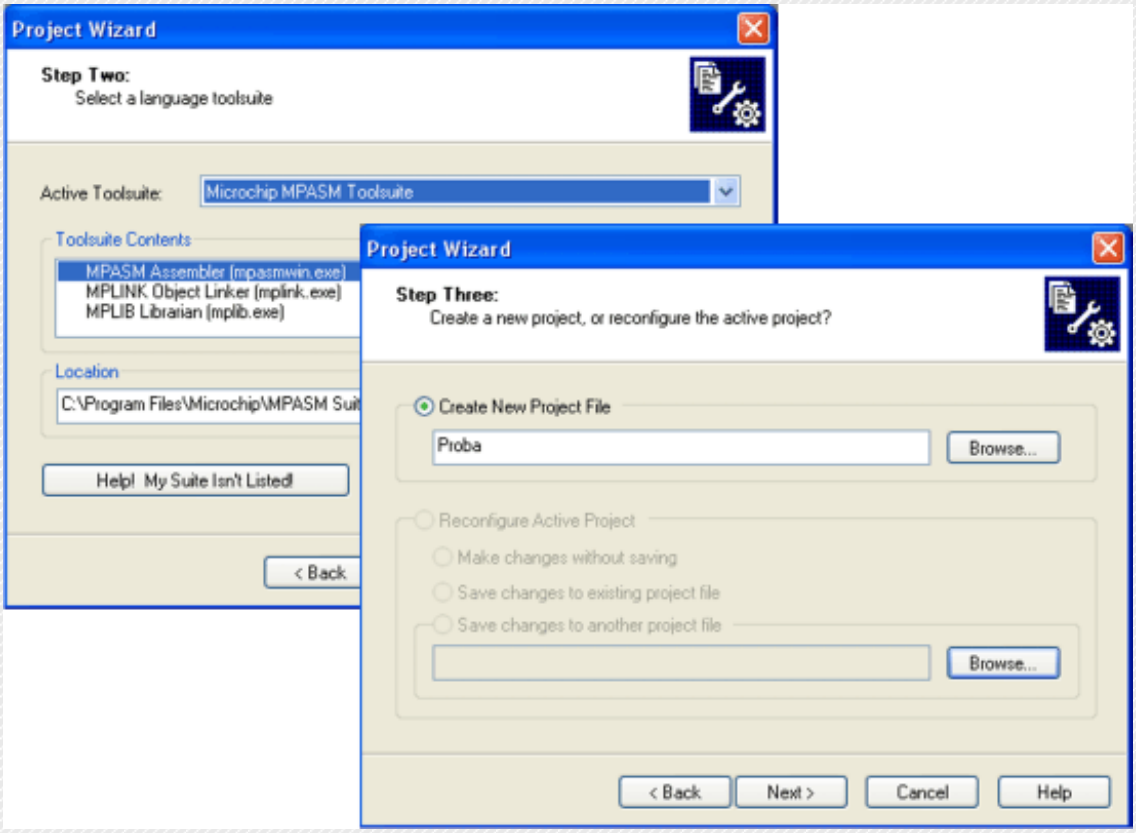




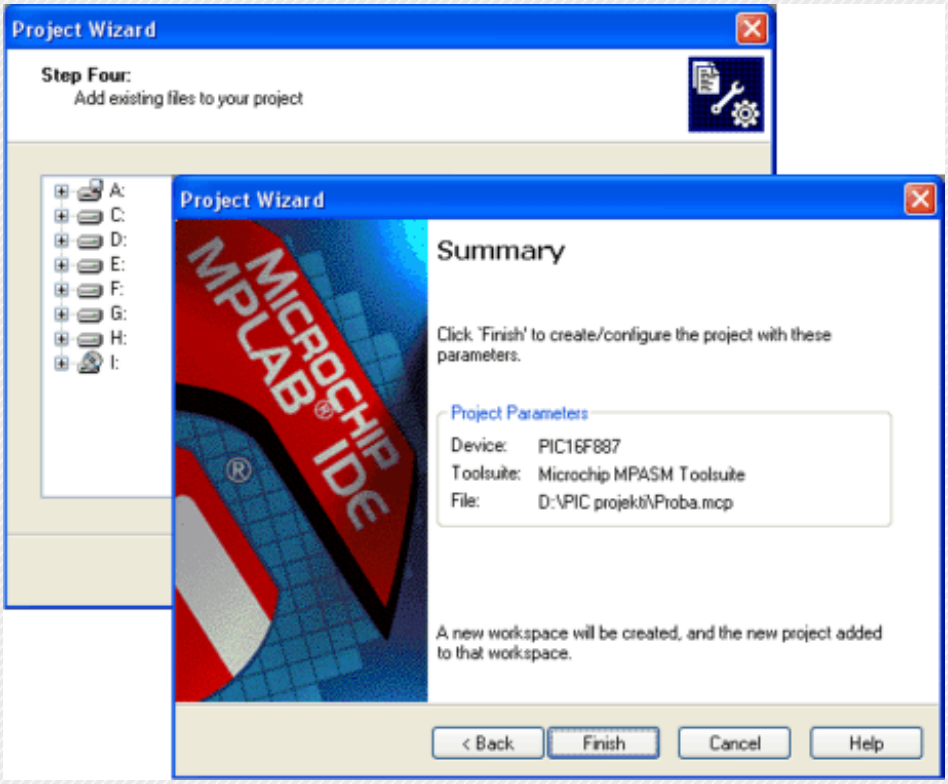
Keep on project-making by clicking NEXT. Then select the microcontroller you will be using.

In our case, it is PIC16F887 microcontroller.

At the end, the project is assigned a name which usually indicates the purpose and the content of the program being written. The project should be moved to the desired folder. It is best that the folder associates with PIC microcontrollers (See figure).



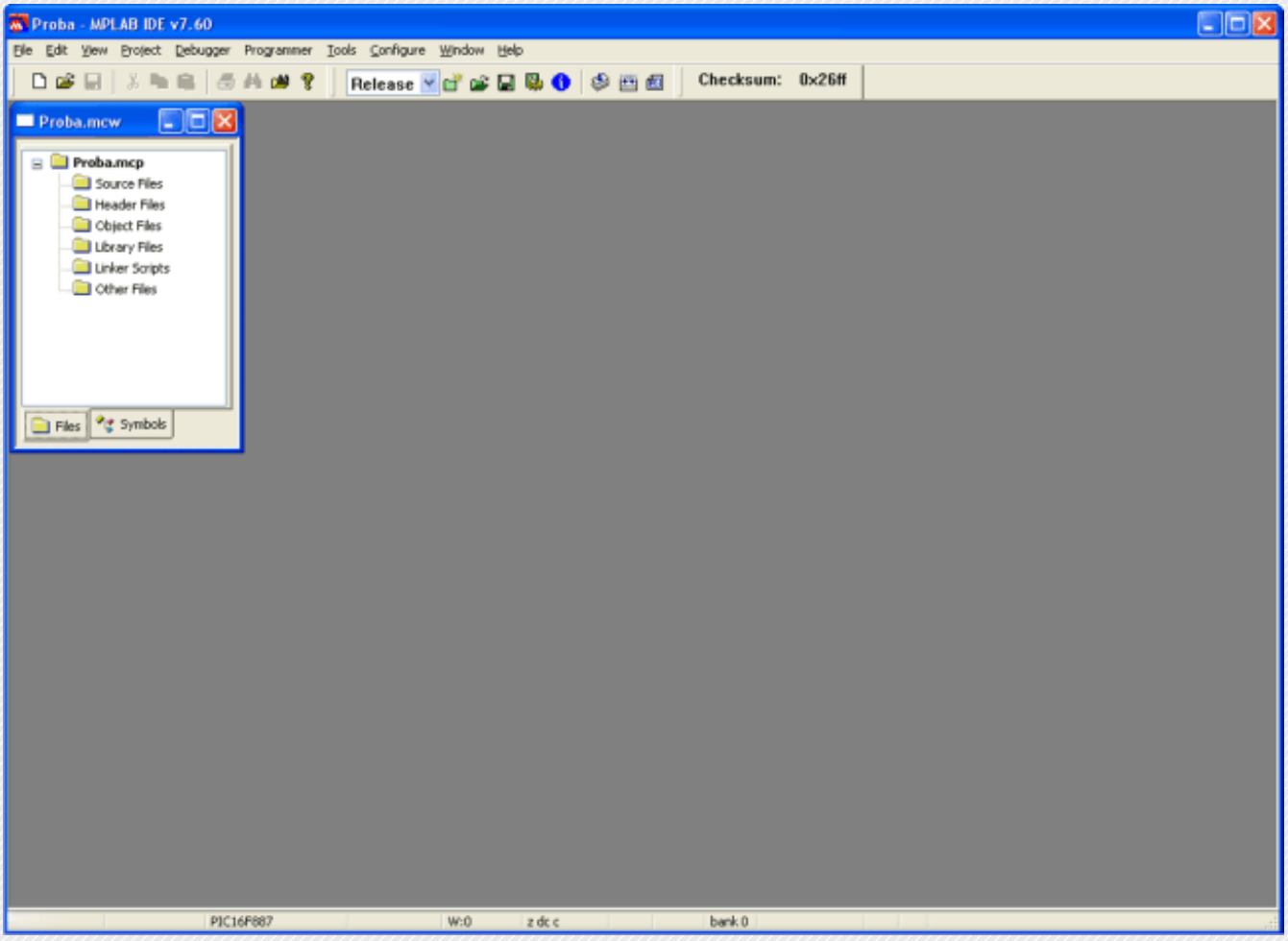
Documents contained in the project do not always need to be written in MPLAB. Documents written by some other program may also be included in the project. In this example, there are no such documents. Just click Next.



Click FINISH to complete the project. The window itself contains project parameters.

WRITING A NEW PROGRAM

When the project is created, a window shown in figure below appears.

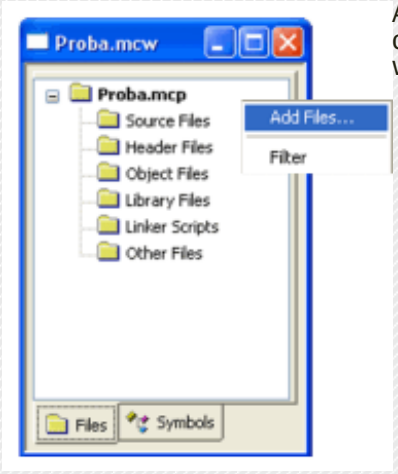


The next step is to write a program. Open a new document by clicking **File>New**. Text Editor in MPLAB environment appears.

Save the document in the folder **D:\PIC projects** by using the **File>Save As** command and name it "Blink.asm" indicating that this program is to be an example of port diode blinking. Obviously you can locate you files wherever you wish, in whichever hard drive you wish. Using a common directory to store all your different projects and subdirectories in makes good sense.

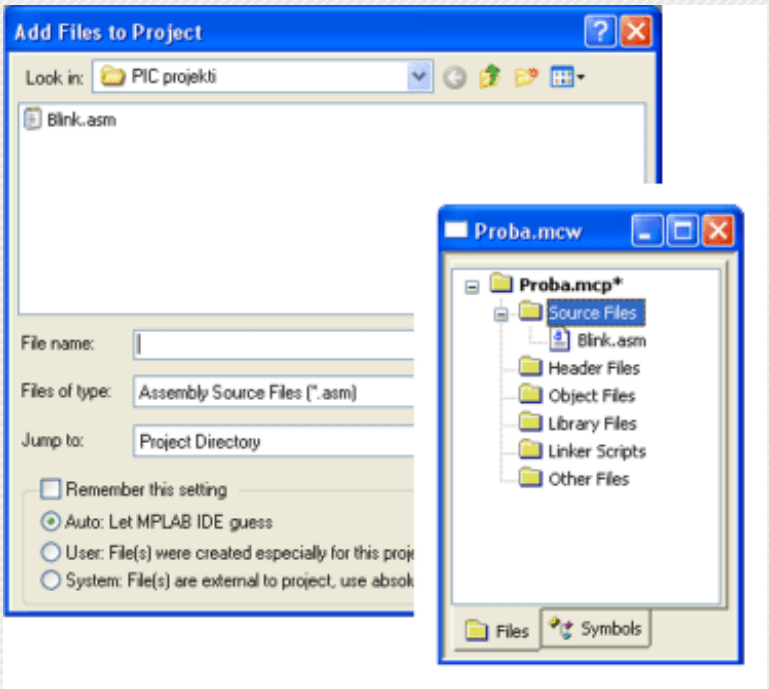
Example:

```
D:\Pic Projects
    LED Flash Project
        All associated files
    Event Count Project
        All associated files
    LED Scanning Project
        All associated files
```



After the "Blink.asm" is created and saved, it should be included in the project by right click on the "Source Files" option in the "Proba.mcw" window. After that, a small window with two options appears. Select the first one "Add Files".

Click on that option opens another window containing the folder PIC along with the document Blink.asm. See figure below.



Click "Blink" to include the document Blink.asm into the project.

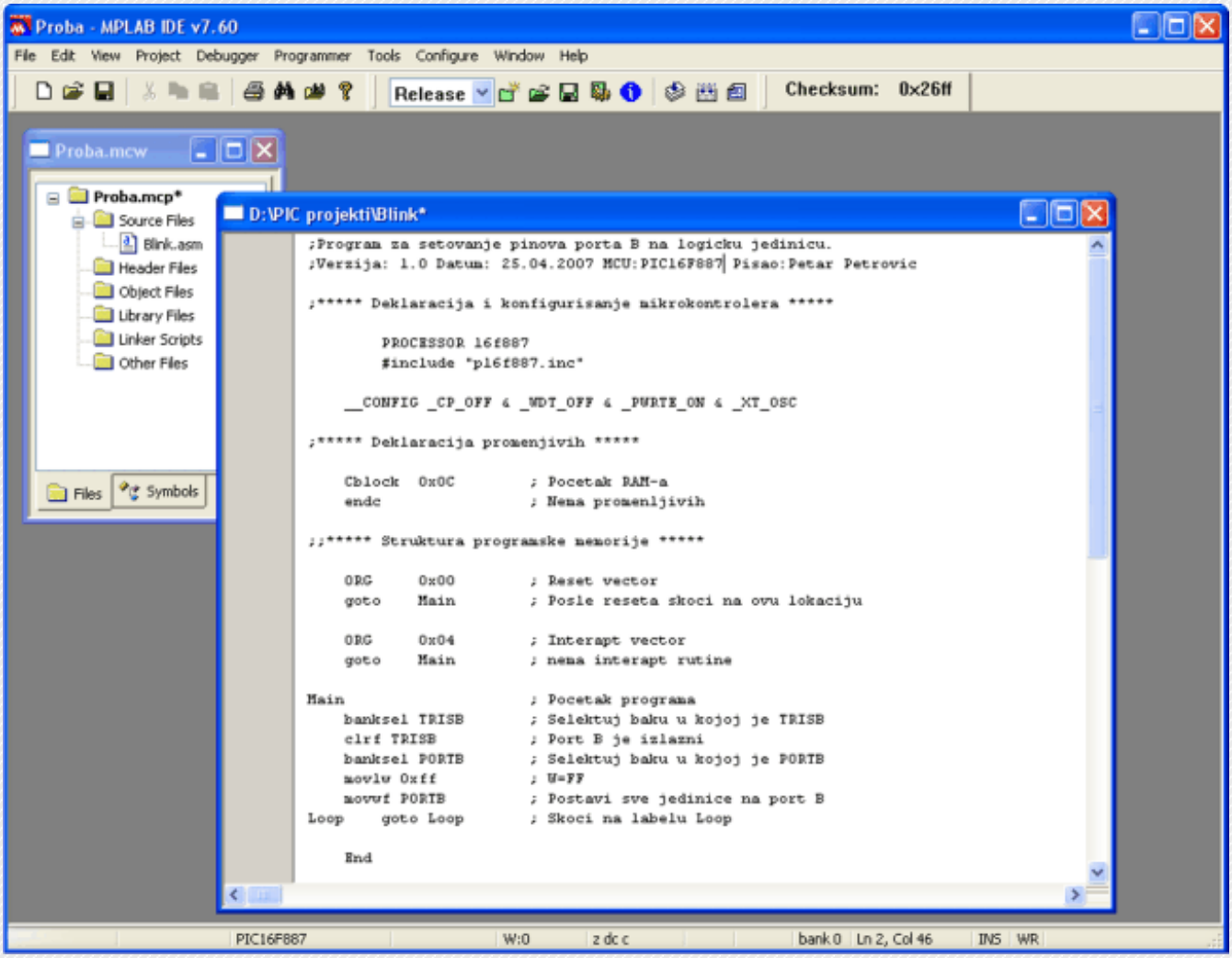
Program writing example

The program writing procedure cannot start until all previous operations have been performed. Program written below is a simple illustration of project-making.

```
;Program to set port B pins to logic one (1).
;Version: 1.0 Date: April 25,2007 MCU: PIC16F887 Programmer: John Smith

;***** Declaration and configuration of the microcontroller *****

        PROCESSOR 16f887
        #include "p16f887.inc"
        __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

;***** Variable declaration *****

        Cblock    0x20                  ; First free RAM location
        endc                            ; No variables

;;***** Program memory structure *****

        ORG       0x00                  ; Reset vector
        goto      Main                  ; After reset jump to this location

        ORG       0x04                  ; Interrupt vector
        goto      Main                  ; No interrupt routine
Main                                    ; Start the program
        banksel   TRISB                 ; Select bank containing TRISB
        clrf      TRISB                 ; Port B is configured as output
        banksel   PORTB                 ; Select bank containing PORTB
        movlw     0xff                  ; W=FF
        movwf     PORTB                 ; Move W to port B
Loop    goto      Loop                  ; Jump to label Loop

        End
```

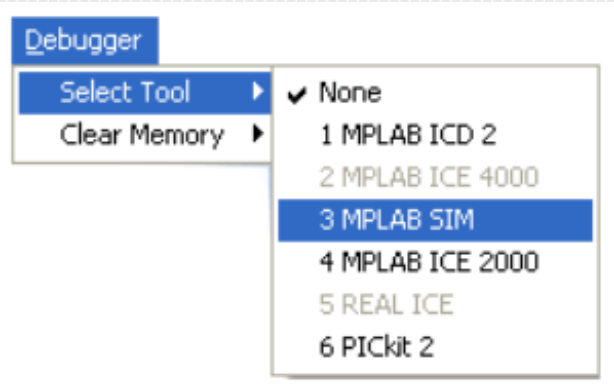
The program should be written to the ‘Blink.asm’ window or copied from disc by means of options *copy/paste*. When copied, the program should be compiled into executable HEX format by using option PROJECT -> BUILD ALL. A new window appears. The last sentence is the most important because it tells us whether compiling has succeeded or not. Clearly, ’**BUILD SUCCEEDED**’ message means that no error occurred and compiling has been successfully done.

In case an error occurs, it is necessary to click twice on the message referring to it in the ‘Output’ window, which automatically switch you over to assembly program, directly to the line where the error has occurred.



SIMULATOR

Asimulator is a part of MPLAB environment which provides better insight into the operation of the microcontroller. Generally speaking, a simulation is an attempt to model a real-life or hypothetical situation so that it can be studied to see how the system works. By means of the simulator, it is also possible to monitor current values of variables, registers and port pins states as well. To be honest, a simulator is not of the same importance for all programs. If a program is simpler (as in our example), the simulation is not of great importance because setting port B pins to logic one (1) is not complicated at all. However, in more complex programs containing timers, different conditions and requests (especially mathematical operations), the simulator may be of great use. As the name itself indicates, a simulation means to simulate the operation of microcontroller. Like the microcontroller, a simulator executes instructions one after another (line by line) and constantly updates the state of all registers. In this way, the user simply monitors program execution. At the end of program writing, the user should first test it in the simulator prior to executing it in a real environment. Unfortunately, this is one of many good things being overlooked by the programmer because of its character as such and the lack of high-quality simulators as well.



Simulator is activated by clicking on **DEBUGGER > SELECT TOOL > MPLAB SIM**, as shown in figure. As a result, several icons related to the simulator only appear. Their meanings are as follows:

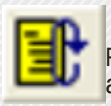

Starts program execution at full speed. In this example, the simulator executes the program at full (normal) speed until it is halted by clicking the icon below.


Pauses program execution. Program can continue executing step by step or at full speed again.

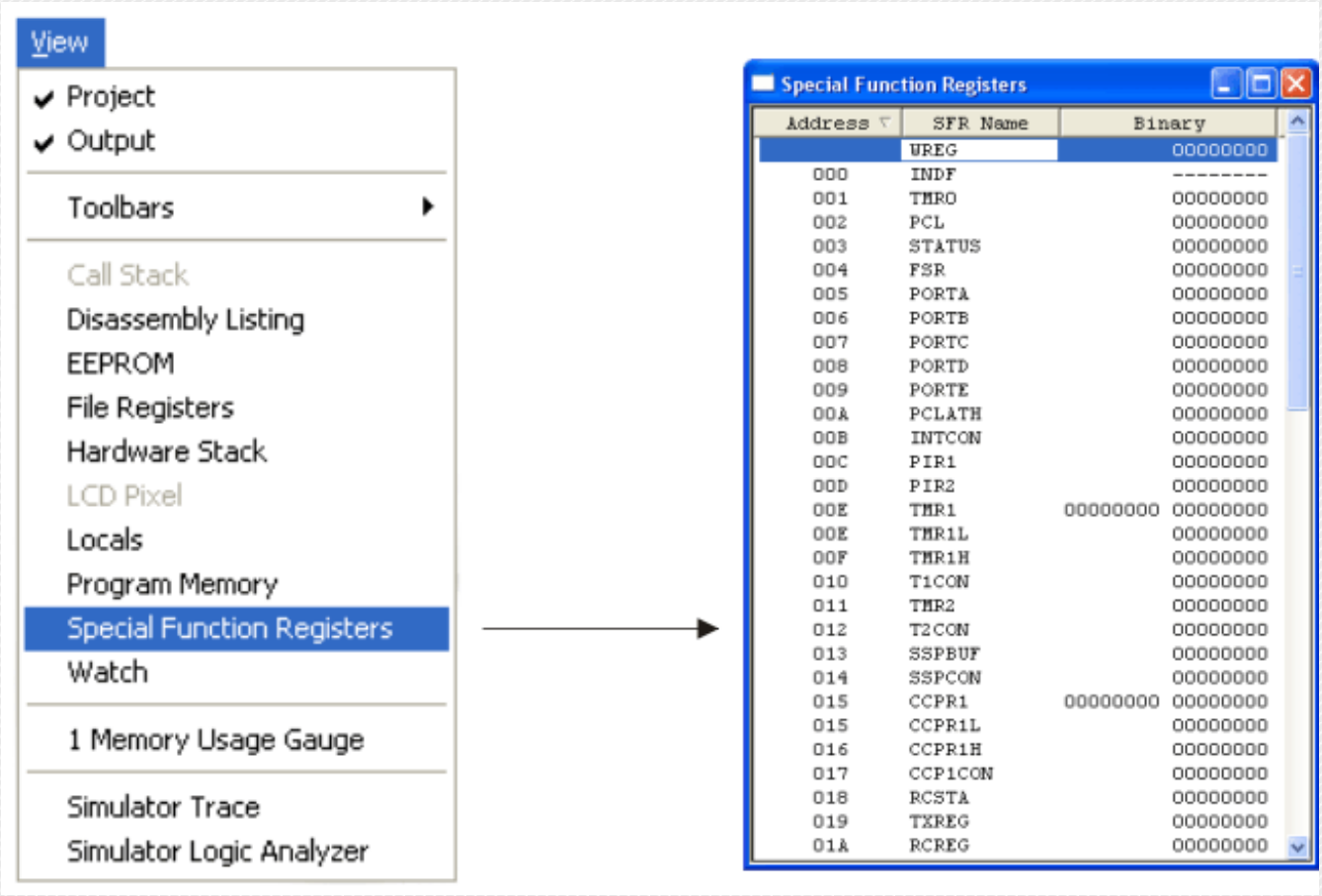

Starts program execution at optional speed. The speed of execution is set in dialog *Debugger/Settings/Animation/Realtime* Updates.


Starts step-by-step program execution. Instructions are executed one after another. Furthermore, clicking on this icon enables you to step into subroutines and macros.


This icon has the same function as the previous one except it has the ability to step into subroutines.


Resets microcontroller. By clicking this icon, the program counter is positioned at the beginning of the program and simulation can start.

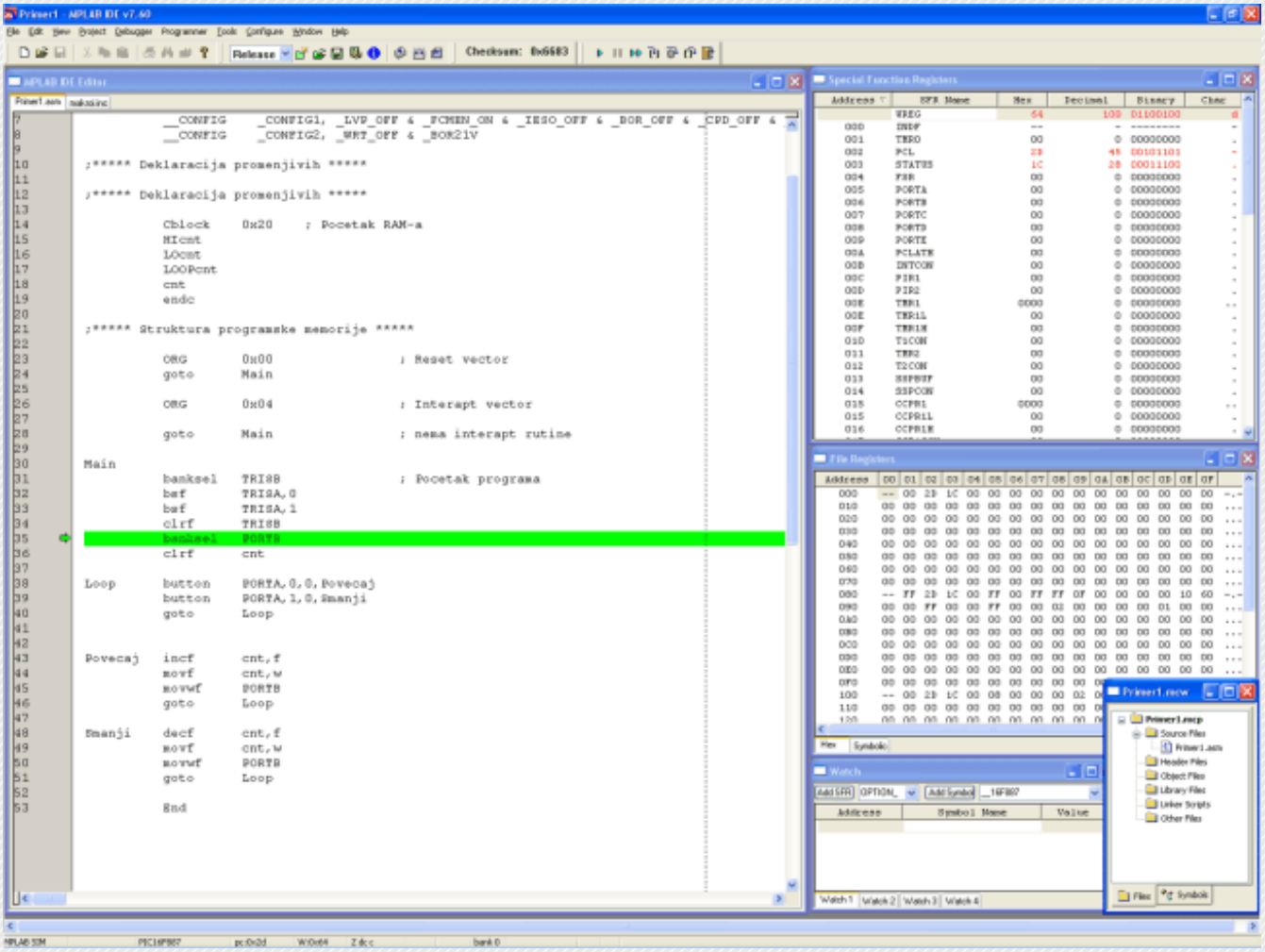
Similar to real environment, the first thing that should be done is to reset the microcontroller using the option DEBUGGER > RESET or by clicking reset icon. As the consequence of this, a green line is positioned at the beginning of the program and program counter PCL is cleared to zero. Refer to the window *Special Function Registers* shown below.



Apart from SFRs, it is good to have an insight in File Registers. A window containing them appears by clicking the **VIEW-**

>FILE REGISTERS option.

If the program contains variables, it is good to monitor their values as well. Each variable is assigned a window (Watch Windows) by clicking **VIEW->WATCH** option.



If the program contains variables, it is good to monitor their values as well. Each variable is assigned a window (*Watch Windows*) by clicking **VIEW->WATCH** option.

After all variables and registers of interest become available on the simulator working area, the process of simulation can start. The next instruction may be either **Step into** or **Step over** depending on whether you want to step into subroutine or not. The same instructions may be set by using keyboard- push-buttons <F7> or <F8> (generally, all important instructions have the corresponding pushbuttons on the keyboard).

Previous Chapter | Table of Contents | Next Chapter

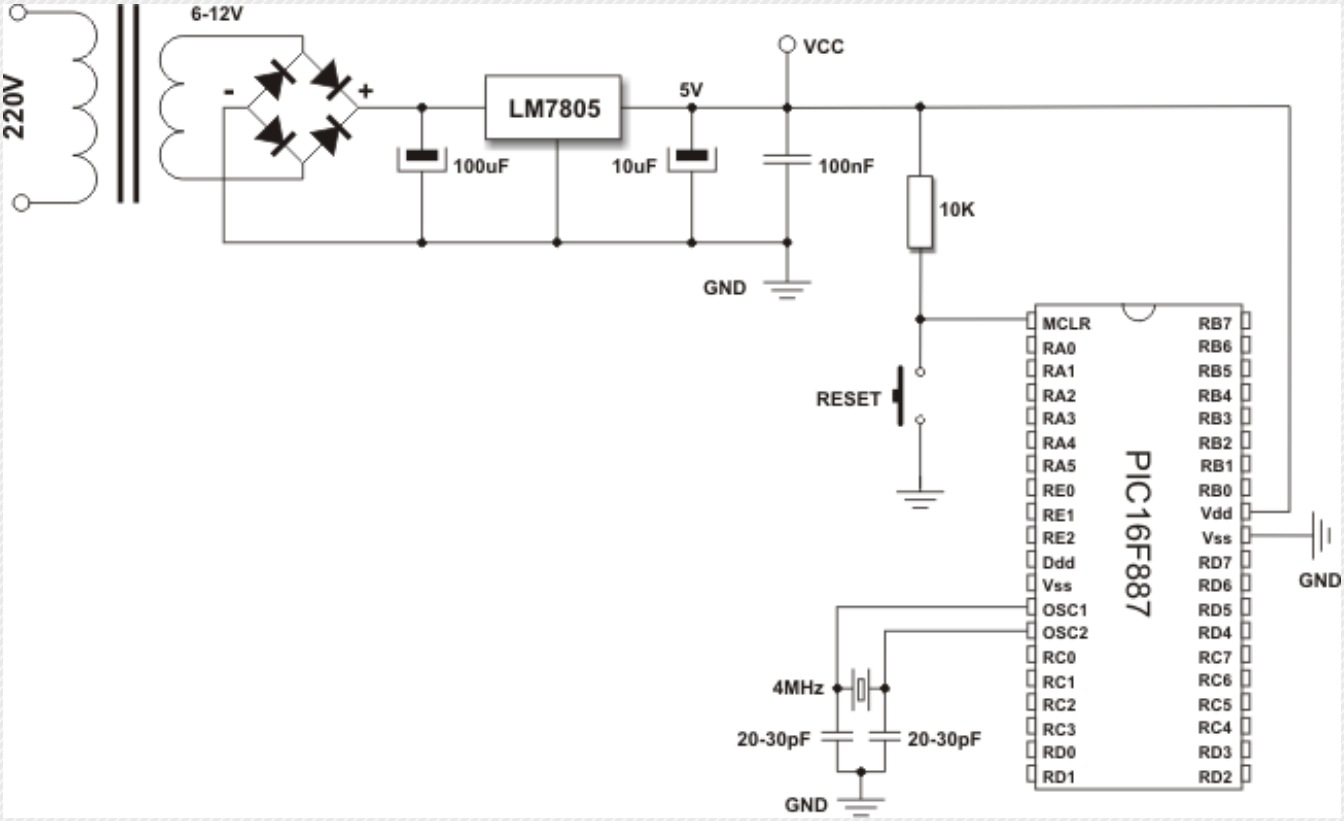
Appendix B: Examples

The purpose of this chapter is to provide basic information about microcontrollers that one needs to know in order to be able to use them successfully in practice. This chapter, therefore, does not contain any super interesting program or device schematic with amazing solutions. Instead, the following examples are more proof that program writing is neither a privilege nor a talent issue but the ability of simply putting puzzle pieces together using directives. Rest assured that design and development of devices mainly consists of the following method "test-correct-repeat". Of course, the more you are in it the more complicated it becomes since the puzzle pieces are put together by both children and first-class architects...

BASIC CONNECTING

As seen in the figure below, in order to enable the microcontroller to operate properly it is necessary to provide:

- Power Supply;
- Reset Signal; and
- Clock Signal.



Clearly, it is about simple circuits, but it does not have to always be like that. If the target device is used for controlling expensive machines or maintaining vital functions, everything gets more and more complicated! However, this solution is sufficient for the time being...

POWER SUPPLY

Even though the PIC16F887 can operate at different supply voltages, why to test "Murphy's low"?! A 5DC power supply is shown above. The circuit, uses a cheap integrated three-terminal positive regulator, LM7805, provides high-quality voltage stability and quite enough current to enable microcontroller and peripheral electronics to operate normally (enough in this case means 1Amp).

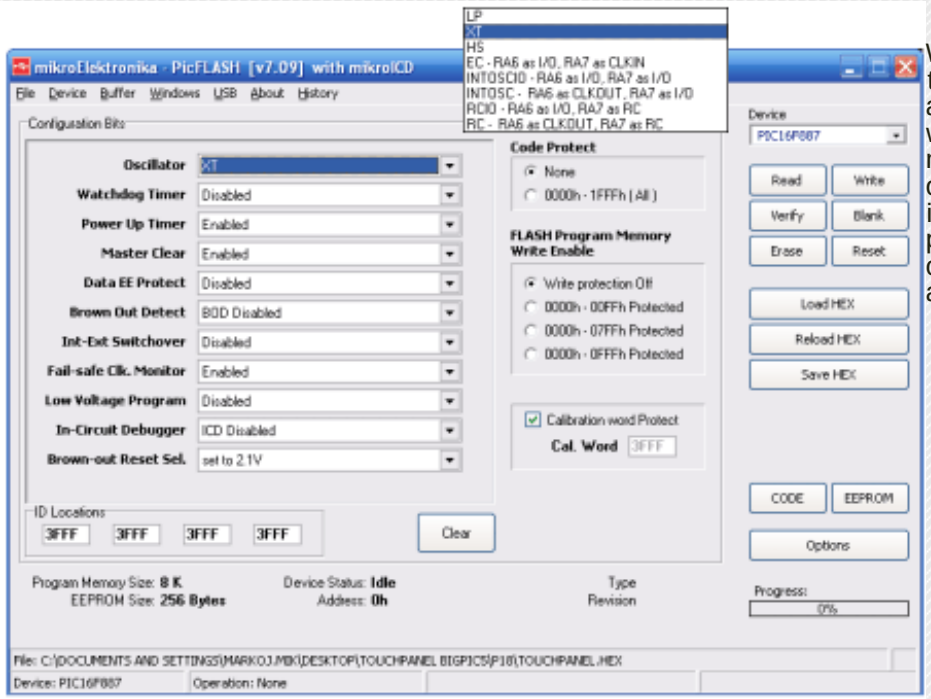
RESET SIGNAL

In order that the microcontroller can operate properly, a logic one (VCC) must be applied on the reset pin it explains the connection pin-resistor 10K-VCC. The push-button connecting the reset pin MCLR to GND is not necessary. However, it is almost always provided because it enables the microcontroller safe return to normal operating conditions if something goes wrong. By pushing this button, 0V is brought to the pin, the microcontroller is reset and program execution starts from the beginning. The 10K resistor is there to allow 0V to be applied to the MCLR pin, via the push-button, without shorting the 5VDC rail to ground.

CLOCK SIGNAL

Even though the microcontroller has a built in oscillator, it cannot operate without external components which stabilize its operation and determine its frequency (operating speed of the microcontroller). Depending on which elements are in use as well as their frequencies, the oscillator can be run in four different modes:

- LP - Low Power Crystal;
- XT - Crystal / Resonator;
- HS - High speed Crystal / Resonator; and
- RC - Resistor / Capacitor.

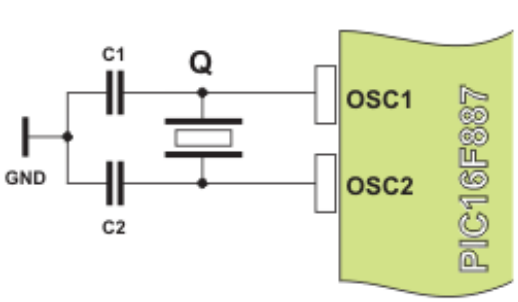


Why are these modes so important? Owing to the fact that it is almost impossible to make a stable oscillator which operates over a wide frequency range, the microcontroller must know which crystal is connected in order that it can adjust the operation of its internal electronics to it. This is why all programs used for chip loading contains an option for oscillator mode selection. See above figure.

Quartz resonator

When a quartz crystal is used for frequency stabilization, the built in oscillator operates at a very precise frequency which is isolated from changes in temperature and voltage power supply as well. This frequency is normally labelled on the microcontroller package.

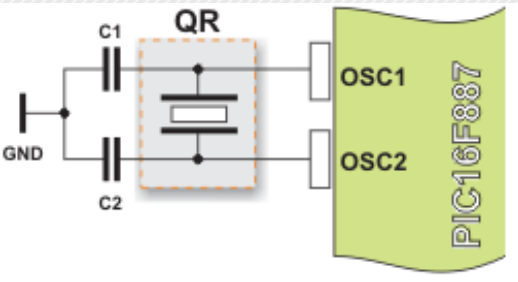
Apart from the crystal, capacitors C1 and C2 must be also connected as per the schematic below. Their capacitance is not of great importance, therefore, the values provided in the table should be considered as a recommendation rather than a strict rule.



Mode	Frequency	C1, C2
LP	32 KHz	33pF
	200 KHz	15pF
XT	200 KHz	47-68 pF
	1 MHz	15 pF
	4 MHz	15 pF
HS	4 MHz	15 pF
	8 MHz	15-33 pF
	20 MHz	15-33 pF

Ceramic resonator

Ceramic resonator is cheaper, but very similar to quartz by its function and the way of operating. This is why the schematics illustrating their connection to the microcontroller are identical. However, the capacitor value is a bit different in this case due to different electric features. Refer to the table.

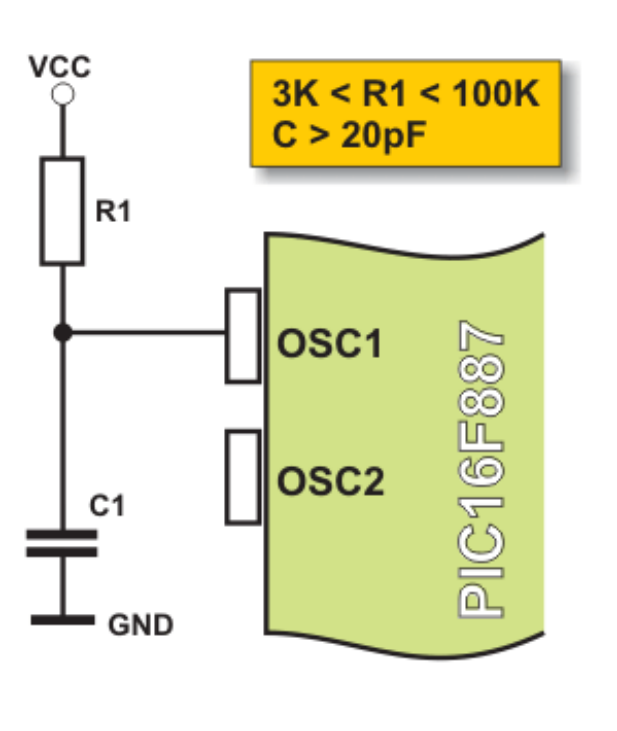


Mode	Frequency	C1, C2
XT	455 KHz	68-100 pF
	2 MHz	15-68 pF
	4 MHz	15-68 pF
HS	8 MHz	10-68 pF
	16 MHz	10-22 pF

These oscillators are used when it is not necessary to have extremely precise frequency.

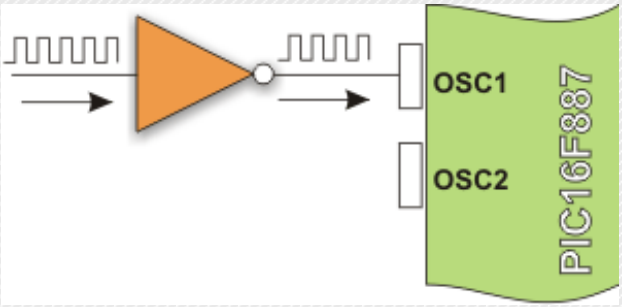
RC oscillator

If the operating frequency is not of importance then there is no need for additional expensive components for stabilization. Instead, a simple RC network, as shown in the figure below, will be enough. Since only the input of the local oscillator input is in use here, the clock signal with frequency Fosc/4 will appear on the OSC2 pin. Furthermore, that frequency becomes operating frequency of the microcontroller, i.e. the speed of instruction execution.



External oscillator

If it is required to synchronize the operation of several microcontrollers or if for some reason it is not possible to use any of the previous schematics, a clock signal may be generated by an external oscillator. Refer to the figure below.



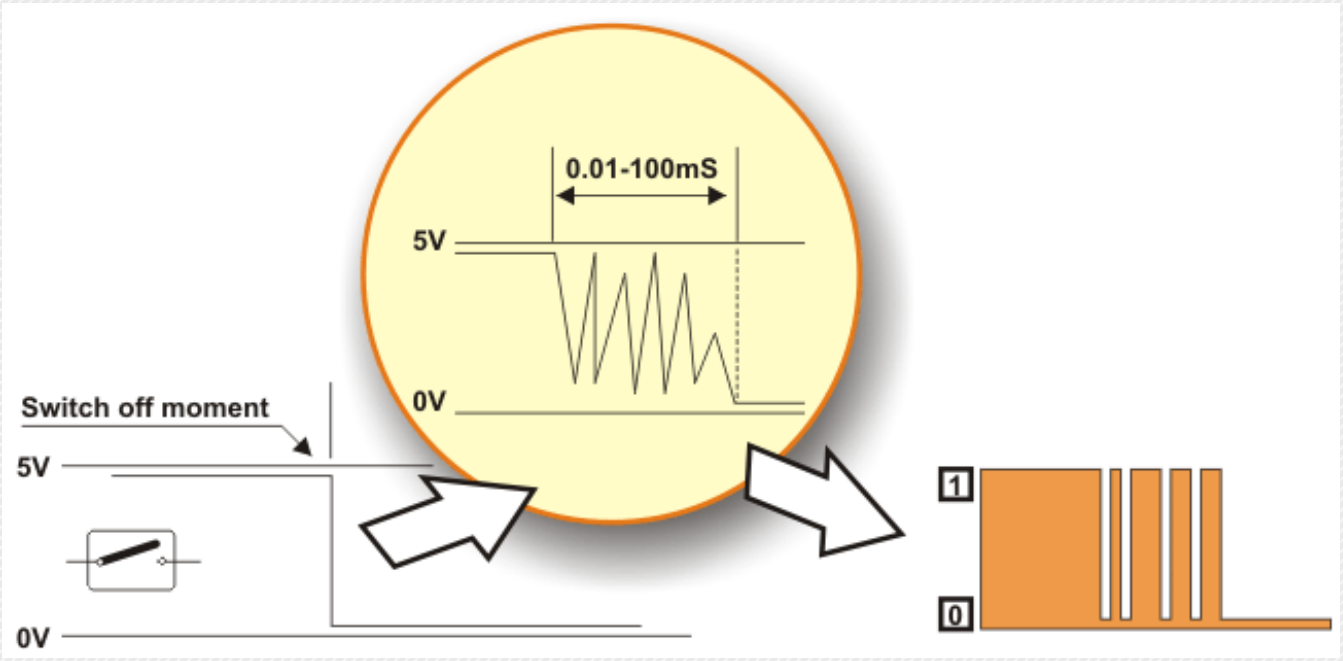
ADDITIONAL COMPONENTS

Regardless of the fact that the microcontroller is a product of modern technology, it is of no use without being connected to additional components. Simply, the appearance of voltage on the microcontroller pins mean nothing if not used for performing certain operations (turn something on/off, shift, display etc.).

This section intentionally covers only the most commonly used additional components in practice such as resistors, transistors, LED diodes, LED displays, LCD displays and RS232 communication circuits.

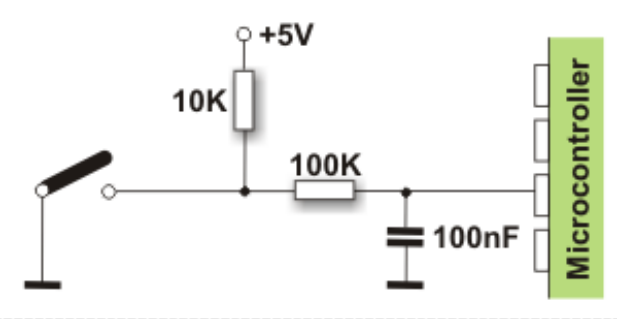
SWITCHES AND PUSH-BUTTONS

There is nothing simpler than switches and push-buttons! This is definitely the simplest way of detecting the appearance of a voltage on the microcontroller input pin and there is no need for additional explanation of how these components operate. Nevertheless, it is not so simple in practice... Then, what is it all about?

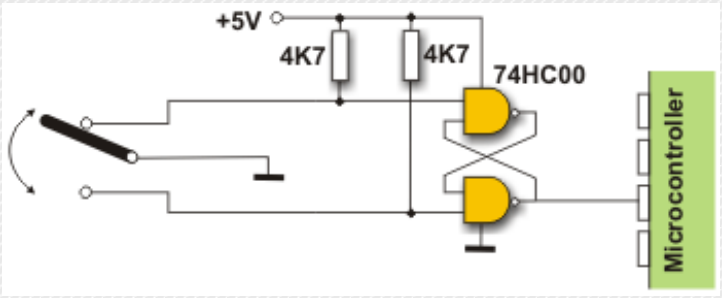


It is about contact bounce- a common problem with mechanical switches. When the contacts strike together, their momentum and elasticity act together to cause bounce. The result is a rapidly pulsed electrical current instead of a clean transition from zero to full current. Generally, it mostly occurs due to vibrations, slight rough spots and dirt between contacts. This effect is usually unnoticeable when using these components in everyday life because the bounce happens too quickly to affect most equipment, but causes problems in some analogue and logic circuits that respond fast enough to misinterpret the on-off pulses as a data stream. Anyway, the whole process does not last long (a few micro- or milliseconds), but long enough to be registered by the microcontroller. When using only a push-button as a pulse counter, errors occurs in almost 100% of cases!

This problem may be easily solved by connecting a simple RC circuit to surpress quick voltage changes. Since the bounce period is not defined, the values of components are not precisely determined. In most cases, it is recommended to use the values as shown in figure below.



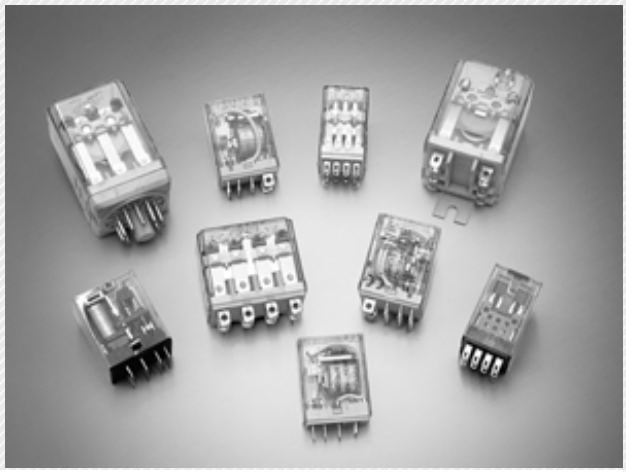
If complete stability is needed then radical measures should be taken! The output of the circuit, shown in figure below (RS flip-flop), will change its logic state only after detecting the first pulse triggered by contact bounce. This solution is more expensive (SPDT switch), but the problem is definitely solved!



In addition to these hardware solutions, there is also a simple software solution. When a program tests the state of an input pin and detects a change, the check should be done one more time after a certain delay. If the program confirms

the change, it means that a switch/push-button has changed its position. The advantages of such solution are obvious: it is free of charge, effects of noises are eliminated and it can be applied to the poorer quality contacts as well.

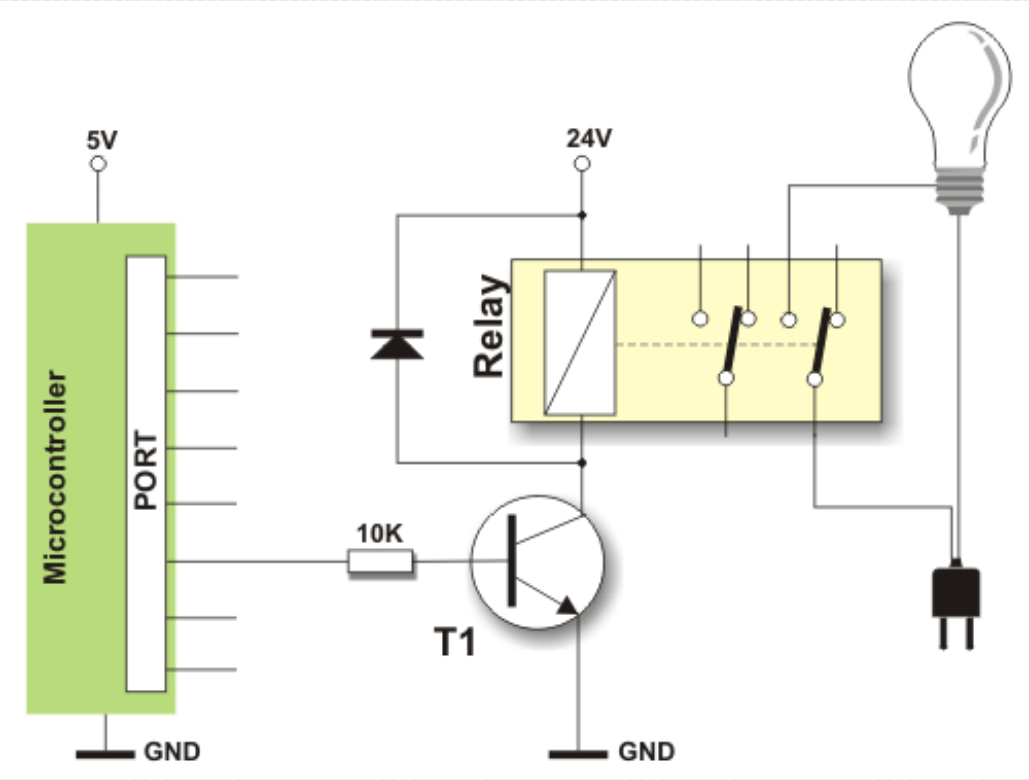
RELAY



A relay is an electrical switch that opens and closes under the control of another electrical circuit. It is therefore connected to output pins of the microcontroller and used to turn on/off high-power devices such as motors, transformers, heaters, bulbs, etc. These devices are almost always placed away from the boards sensitive components. There are various types of relays, but all of them operate in the same way. When a current flows through the coil, the relay is operated by an electromagnet to open or close one or many sets of contacts. Similar to optocouplers, there is no galvanic connection (electrical contact) between input and output circuits. Relays usually demand both higher voltage and current to start operation but there are also miniature ones that can be activated by a low current directly obtained from a microcontroller pin.

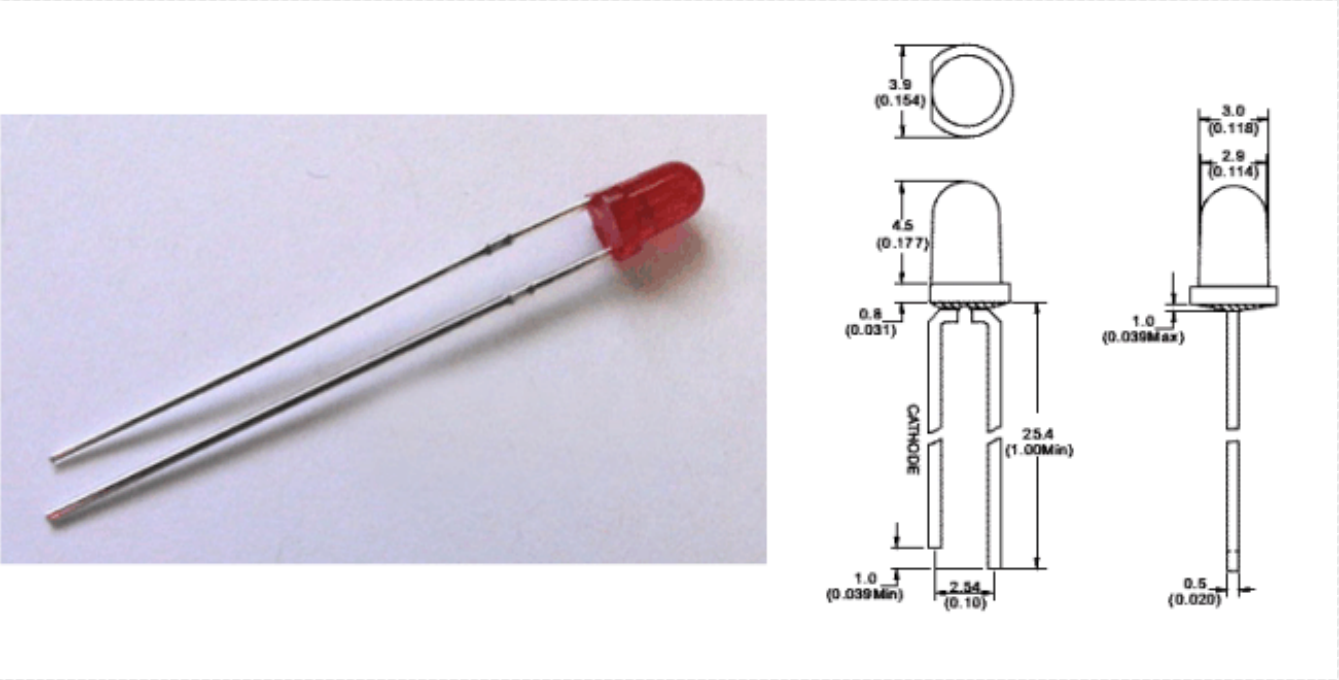
This figure shows the most commonly used solution.

In order to prevent the appearance of high voltage self-induction caused by a sudden stop of current flow through the coil, an inverted polarized diode is connected in parallel to the coil. The purpose of this diode is to "cut off" the voltage peak.



LED DIODES

You probably know all you need to know about LED diodes, but we should also think of the younger generations...How to destroy a LED?! Well...Very simple.



Quick burning

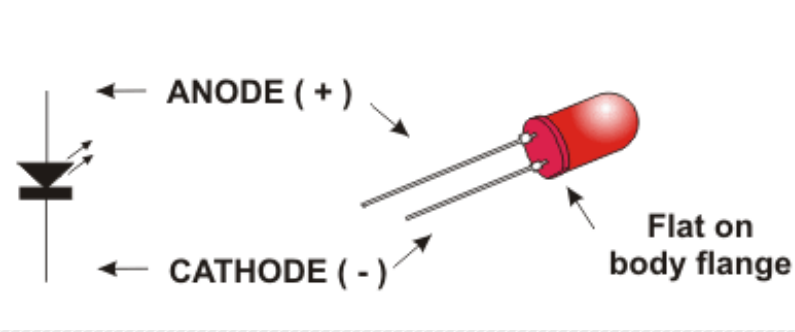
Like any other diode, LEDs have two ends an anode and a cathode. Connect it properly to a power supply voltage. The diode will happily emit light. Turn it upside down and apply the same power supply voltage (even for a moment). It will not emit light- NEVER AGAIN!

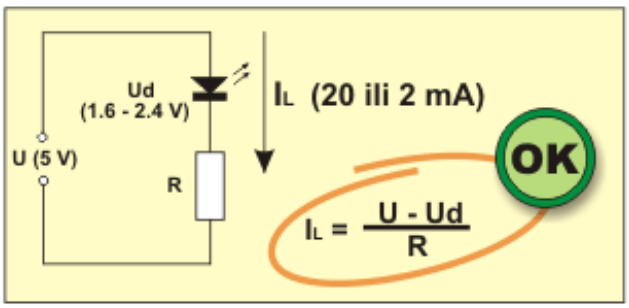
Slow burning

There is a nominal, i.e. maximum current determined for every LED which should not be exceeded. If it happens, the diode will emit more intensive light, but not for a long time!

Something to remember

Similar to the previous example, all you need to do is to discard a current limiting resistor shown below. Depending on power supply voltage, the effect might be spectacular!





LED DISPLAY

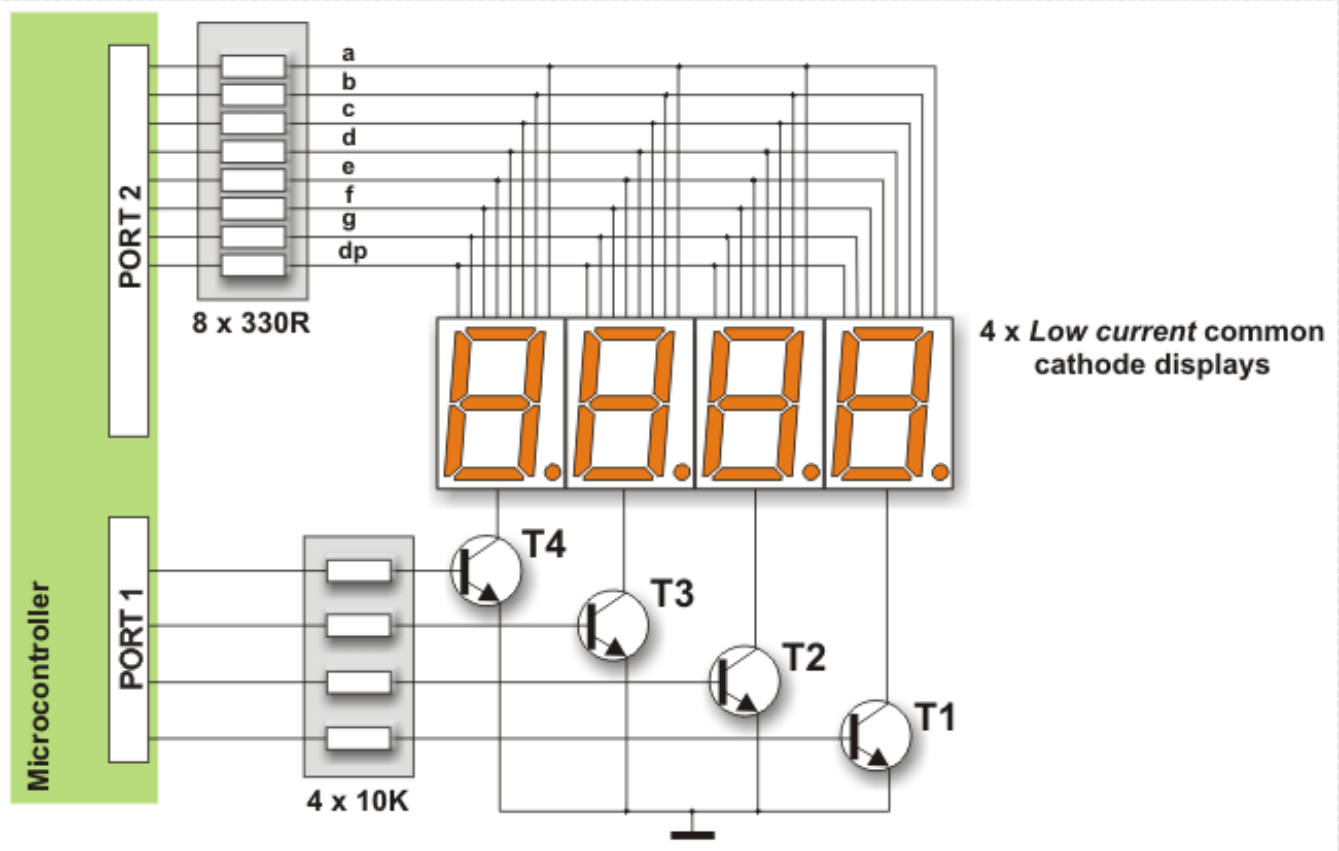
Basically, LED display is nothing more than several LEDs moulded in the same plastic case. There are many types of displays composed of several dozens of built in diodes which can display different symbols. The most commonly used is so called 7-segment display. It is composed of 8 LEDs- 7 segments are arranged as a rectangle for symbol displaying and there is an additional segment for decimal point displaying. In order to simplify connection, anodes or cathodes of all diodes are connected to the common pin so that there are common anode displays and common cathode displays, respectively. Segments are marked with the letters from a to g, plus dp, as shown in figure below. On connecting, each diode is treated separately, which means that each must have its own current limiting resistor.

Here are a few important things that one should pay attention to when buying LED displays:

- Depending on whether anodes or cathodes are connected to the common pin, there are common anode displays and common cathode displays. The figure above shows a common anode display. Looking at physical features, there is no difference between these displays at all so it is recommended to check carefully prior installation which of them is in use;
- For each microcontroller pin, there is a maximum current limitation it can receive or give. Because of this, if several displays are connected to the microcontroller it is rec ommended to use so called *Low current* LEDs using only 2mA for operation; and
- Display segments are usually marked with the letters from a to g, but there is no fast rule indicating to which micro controller pins they should be connected. For this reason it is very important to check connecting prior to commencing program writing or designing a device.

Displays connected to the microcontroller usually occupy a large number of valuable I/O pins, which can be a big problem especially when it is needed to display multi-digital numbers. The problem is more than obvious if, for example, it is needed to display two 6-digit numbers (a simple calculation shows that 96 output pins are needed in this case)! This problem has a solution called MULTIPLEXING.

Here is how an optical illusion based on the same operating principle as a film camera is made. Only one digit at a time is active, but they change their state so quickly that one gets impression that all digits of a number are active simultaneously.

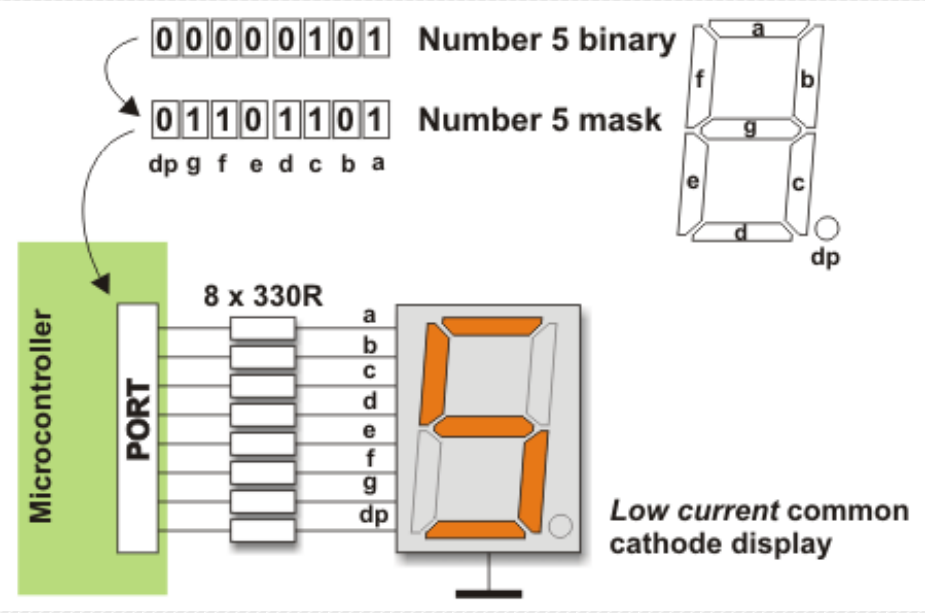


Here is an explanation on the figure above. First a byte representing units is applied on a microcontroller port and a transistor T1 is activated simultaneously. After a while, the transistor T1 is turned off, a byte representing tens is applied on a port and transistor T2 is activated. This process is being cyclically repeated at high speed for all digits and corresponding transistors.

A disappointing fact which indicates that the microcontroller is just a kind of miniature computer designed to understand only the language of zeros and ones is fully expressed when displaying any digit. Namely, the microcontroller does not know what units, tens or hundreds are, nor what ten digits we are used to look like. Therefore, each number to be displayed must go through the following procedure:

First of all, in a particular subroutine a multi-digital number must be split into units, tens etc. Then, these must be stored in special bytes each. Digits get recognizable format by performing "masking". In other words, a binary format of each digit is replaced by a different combination of bits using a simple subroutine. For example, the digit 8 (0000 1000) is replaced by binary number 0111 1111 in order to activate all LEDs displaying digit 8. The only diode remaining inactive in this case is reserved for the decimal point.

If a microcontroller port is connected to the display in a way that bit 0 activates segment "a", bit 1 activates segment "b", bit 2 segment "c" etc., then the table below shows the mask for each digit.



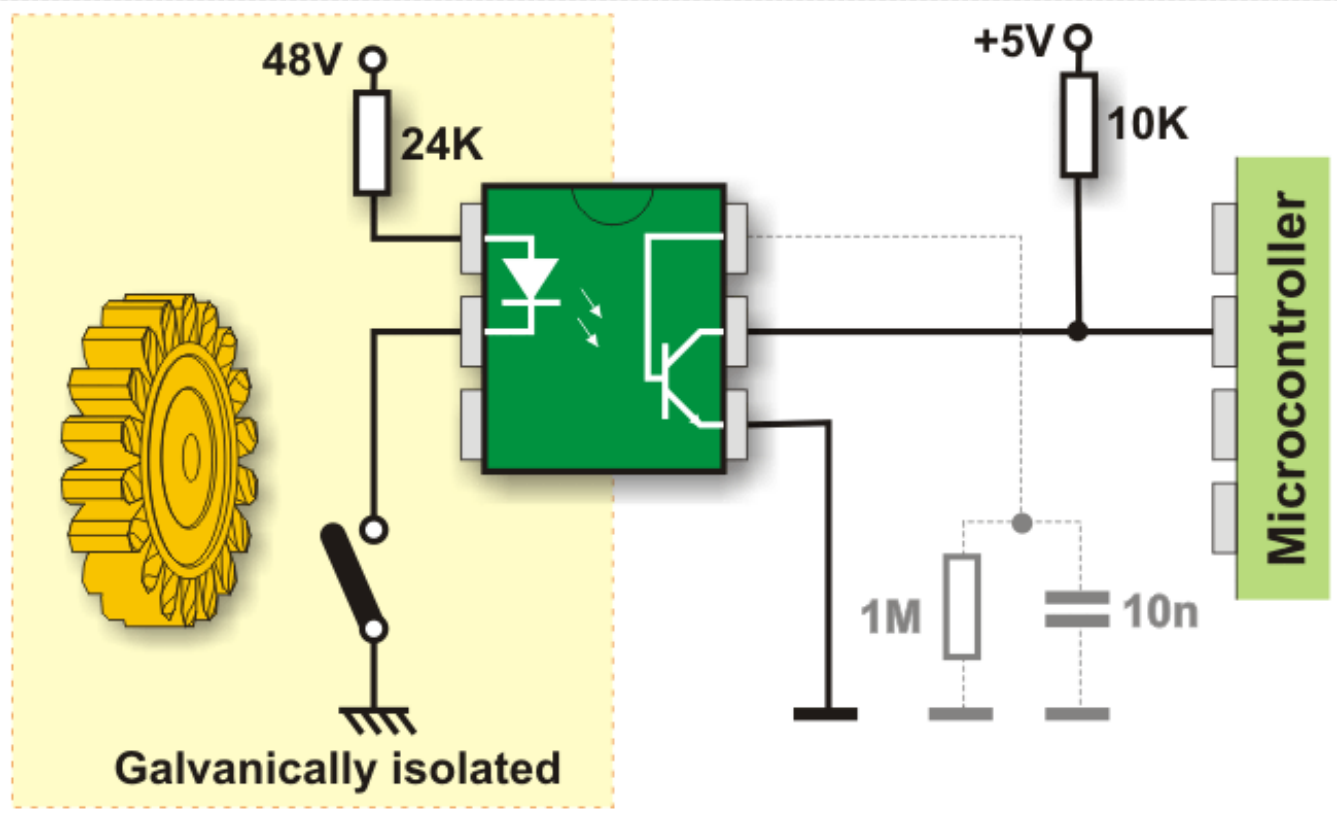
Digits to display	Display Segments							
	dp	a	b	c	d	e	f	g
0	0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0	0
2	0	1	1	0	1	1	0	1
3	0	1	1	1	1	0	0	1
4	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1
6	0	1	0	1	1	1	1	1
7	0	1	1	1	0	0	0	0
8	0	1	1	1	1	1	1	1
9	0	1	1	1	1	0	1	1

In addition to digits from 0 to 9, there are some letters- A, C, E, J, F, U, H, L, b, c, d, o, r, t- that can be also displayed by means of the appropriate masking.

In the event that the common anode displays are used, all ones contained in the previous table should be replaced by zeros and vice versa. Additionally, NPN transistors should be used as drivers as well.

OPTOCOUPLER

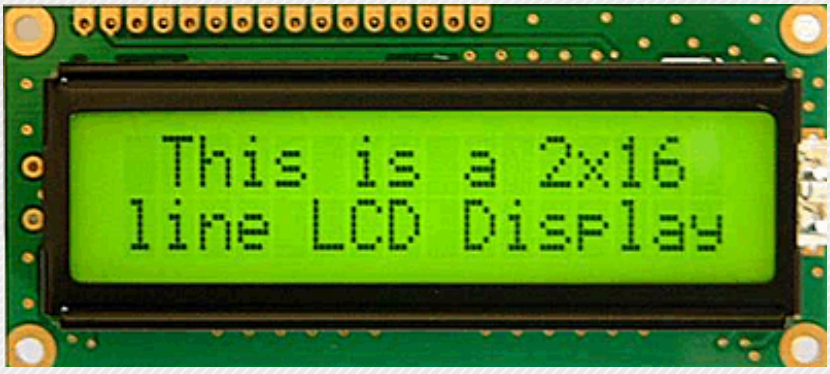
An optocoupler is a device commonly used to galvanically separate microcontroller electronics from any potentially dangerous current or voltage in its surroundings. Optocouplers usually have one, two or four light sources (LED diodes) on their input while on their output, opposite to diodes, there is the same number of elements sensitive to light (phototransistors, photo-thyristors or photo-triacs). The point is that an optocoupler uses a short optical transmission path to transfer a signal between elements of circuit, while keeping them electrically isolated. This isolation makes sense only if diodes and photo-sensitive elements are separately powered. In this way, the microcontroller and expensive additional electronics are completely protected from high voltage and noises which are the most common cause of destroying, damaging or unstable operation of electronic devices in practice. The most frequently used optocouplers are those with phototransistors on their outputs. Additionally, optocouplers with internal base-to-pin 6 connection (there are also optocouplers without it), the base may be left unconnected.



The R/C network represented by the broken line in the figure above denotes optional connection which lessens the effects of noises by eliminating very short pulses.

LCD DISPLAY

This component is specifically manufactured to be used with microcontrollers, which means that it cannot be activated by standard IC circuits. It is used for displaying different messages on a miniature liquid crystal display. The model described here is for its low price and great capabilities most frequently used in practice. It is based on the HD44780 microcontroller (*Hitachi*) and can display messages in two lines with 16 characters each. It displays all the letters of alphabet, Greek letters, punctuation marks, mathematical symbols etc. In addition, it is possible to display symbols made up by the user. Other useful features include automatic message shift (left and right), cursor appearance, LED backlight etc.



LCD DISPLAY

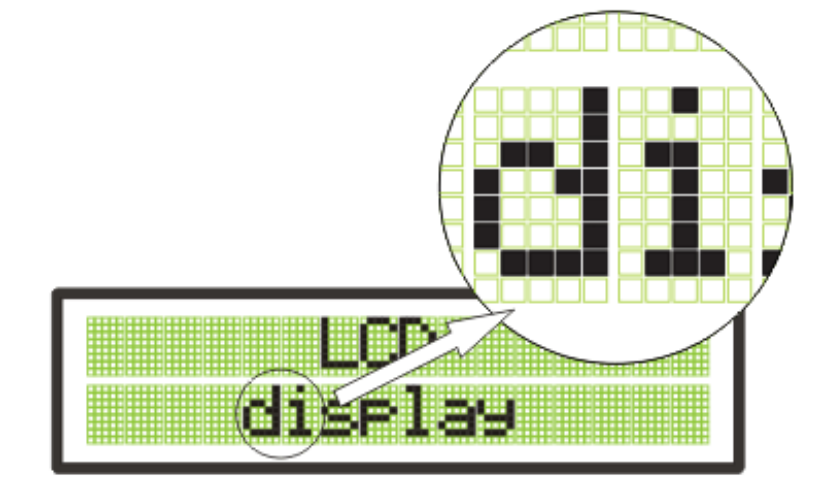
Along one side of a small printed board there are pins used for connecting to the microcontroller. There are in total of 14 pins marked with numbers (16 if the backlight is built in). Their function is described in the table bellow:

Function	Pin Number	Name	Logic State	Description
Ground	1	Vss	-	0V
Power supply	2	Vdd	-	+5V
Contrast	3	Vee	-	0 - Vdd

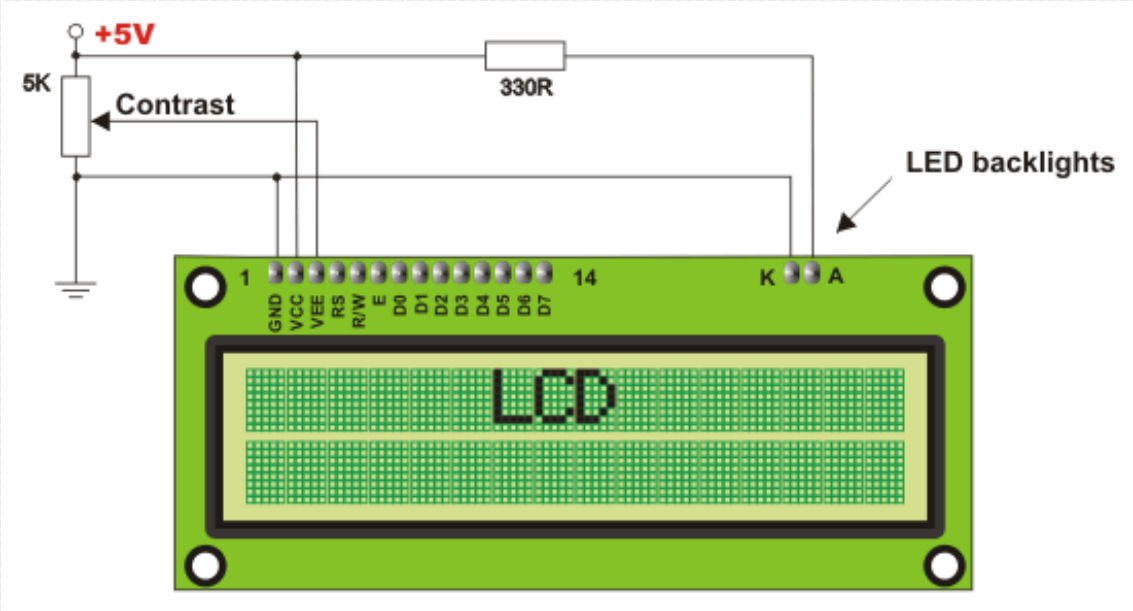
Control of operating	4	RS	0 1	D0 – D7 are interpreted as commands D0 – D7 are interpreted as data
	5	R/W	0 1	Write data (from controller to LCD) Read data (from LCD to controller)
	6	E	0 1 From 1 to 0	Access to LCD disabled Normal operating Data/commands are transferred to LCD
Data / commands	7	D0	0/1	Bit 0 LSB
	8	D1	0/1	Bit 1
	9	D2	0/1	Bit 2
	10	D3	0/1	Bit 3
	11	D4	0/1	Bit 4
	12	D5	0/1	Bit 5
	13	D6	0/1	Bit 6
	14	D7	0/1	Bit 7 MSB

LCD screen

The LCD screen consists of two lines with 16 characters each. Every character consists of 5x8 or 5x11 dot matrix. This book covers the 5x8 character display, which is indeed the most commonly used.



Display contrast depends on the power supply voltage and whether messages are displayed in one or two lines. For this reason, varying voltage 0-Vdd is applied on the pin marked as Vee. Trimmer potentiometer is usually used for that purpose. Some LCD displays have built in backlight (blue or green diodes). When used during operation, a current limiting resistor should be serially connected to one of the pins for backlight (similar to LED diodes).



If there are no characters displayed or if all of them are dimmed when the display is switched on, the first thing that should be done is to check the potentiometer for contrast adjustment. Is it properly adjusted? The same applies if the mode of operation has been changed (writing in one or two lines).

LCD Memory

LCD display contains three memory blocks:

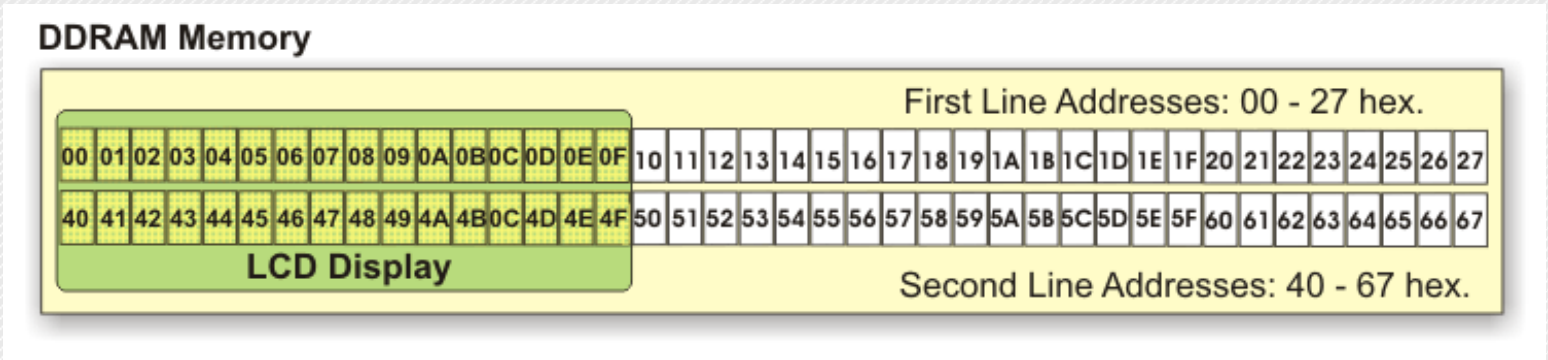
- DDRAM - Display Data RAM;
- CGRAM - Character Generator RAM; and
- CGROM - Character Generator ROM.

DDRAM Memory

DDRAM memory is used for storing characters that should be displayed. The size of this memory is sufficient for storing 80 characters. Some memory locations are directly connected to the characters on display.

It works quite simply: it is enough to configure the display to increment addresses automatically (shift right) and set the starting address for the message that should be displayed (for example 00 hex).

After that, all characters sent through lines D0-D7 will be displayed as a message format we are used to- from left to right. In this very case, displaying starts from the first field of the first line because the address is 00 hex. If more than 16 characters are sent then all of them will be memorized, but only the first sixteen characters will be visible. In order to display the rest of them, a shift command should be used. Virtually, everything looks as if the LCD display is a window which shifts left-right over memory locations containing different characters. In reality, this is how the effect of message shifting on the screen has been created.



If the cursor is on, it appears at the location which is currently addressed. In other words, when a character appears at the cursor position, it will automatically move to the next addressed location.

This is a sort of RAM memory so data can be written to and read from it, but its contents is irretrievably lost upon the power goes off.

CGROM Memory

CGROM memory contains the default character map with all characters that can be displayed on the screen. Each character is assigned to one memory location:

The addresses of CGROM memory locations match the characters of ASCII. If the program being currently executed encounters a command "send character P to port" then the binary value 0101 0000 appears on the port. This value is the ASCII equivalent to the character P. It is then written to LCD, which results in displaying the symbol from the 0101 0000 location of CGROM. In other words, the character "P" is displayed. This applies to all letters of the alphabet (capitals and small), but not to the numbers!

As seen on the previous map, addresses of all digits are pushed forward by 48 relative to their values (digit 0 address is 48, digit 1 address is 49, digit 2 address is 50 etc.). Accordingly, in order to display digits correctly it is necessary to add a decimal number 48 to each of them prior to sending them to LCD.

What is ASCII? From their inception till today, computers can recognize only numbers, but not letters. It means that all data a computer swaps with a peripheral device has a binary format even though the same is recognized by the man as letters (The keyboard is an excellent example)! It's as simple as that- every character matches the unique combination of zeroes and ones. ASCII is character encoding based on the English alphabet. ASCII code specifies a correspondence between standard character symbols and their numerical equivalents.

LCD Basic Commands

All data transferred to LCD through the outputs D0-D7 will be interpreted as a command or a data, which depends on the pin RS logic state:

RS = 1 - Bits D0 - D7 are addresses of the characters to be displayed. LCD processor addresses one character from the character map and displays it. The DDRAM address specifies the location on which the character is to be displayed. This address is defined prior character transfer or the address of the previously transferred character is automatically incremented.

RS = 0 - Bits D0 - D7 are commands which determine display mode.

The commands recognized by the LCD are listed in table below:

Command	RS	RW	D7	D6	D5	D4	D3	D2	D1	D0	Execution Time
Clear display	0	0	0	0	0	0	0	0	0	1	1.64mS
Cursor home	0	0	0	0	0	0	0	0	1	x	1.64mS
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	40uS
Display on/off control	0	0	0	0	0	0	1	D	U	B	40uS
Cursor/Display Shift	0	0	0	0	0	1	D/C	R/L	x	x	40uS
Function set	0	0	0	0	1	DL	N	F	x	x	40uS
Set CGRAM address	0	0	0	1	CGRAM address						40uS
Set DDRAM address	0	0	1	DDRAM address							40uS
Read "BUSY" flag (BF)	0	1	BF	DDRAM address							-
Write to CGRAM or DDRAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	40uS
Read from CGRAM or DDRAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	40uS

```
I/D 1 = Increment (by 1)          R/L 1 = Shift right
    0 = Decrement (by 1)              0 = Shift left

S 1 = Display shift on            DL 1 = 8-bit interface
  0 = Display shift off              0 = 4-bit interface

D 1 = Display on                  N 1 = Display in two lines
  0 = Display off                   0 = Display in one line

U 1 = Cursor on                   F 1 = Character format 5x10 dots
  0 = Cursor off                    0 = Character format 5x7 dots

B 1 = Cursor blink on             D/C 1 = Display shift
  0 = Cursor blink off                0 = Cursor shift
```

What is Busy flag ?

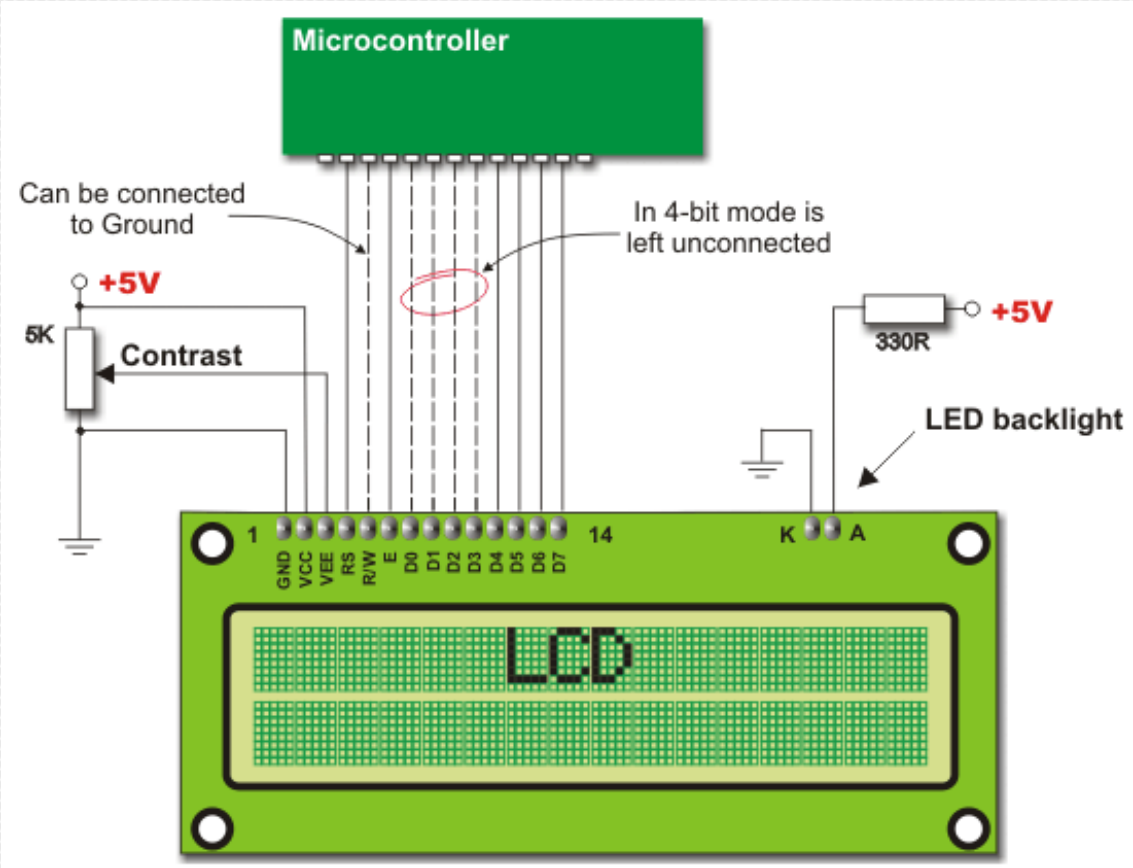
Compared to the microcontroller, the LCD is an extremely slow component. Because of this, it was necessary to provide a

signal which would, upon command execution, indicate that the display is ready for the next piece of data. That signal, called the *busy flag*, can be read from the line D7. When the voltage on this line is 0V (BF=0), the display is ready to receive new data.

LCD Connecting

Depending on how many lines are used for connecting the LCD to the microcontroller, there are 8-bit and 4-bit LCD modes. The appropriate mode is selected at the beginning of the operation in this process called "initialization". 8-bit LCD mode uses outputs D0-D7 to transfer data as explained on the previous page.

The main purpose of 4-bit LED mode is to save valuable I/O pins of the microcontroller. Only 4 higher bits (D4-D7) are used for communication, while others may be unconnected. Each piece of data is sent to the LCD in two steps- four higher bits are sent first (normally through the lines D4-D7) and four lower bits are sent afterwards. Initialization enables the LCD to link and interpret received bits correctly.



Data is rarely read from the LCD (it is mainly transferred from the microcontroller to LCD) so it is often possible to save an extra I/O pin by simple connecting R/W pin to the Ground. Such saving has its price. Messages will be normally displayed, but it will not be possible to read the busy flag since it is not possible to read the display as well. Fortunately, there is a simple solution. After sending a character or a command it is important to give the LCD enough time to do its job. Owing to the fact that the execution of the slowest command lasts for approximately 1.64mS, it will be sufficient to wait approximately 2mS for LCD.

LCD Initialization

The LCD is automatically cleared when powered up. It lasts for approximately 15mS. After this, display is ready for operation. The mode of operation is set by default. It means that:

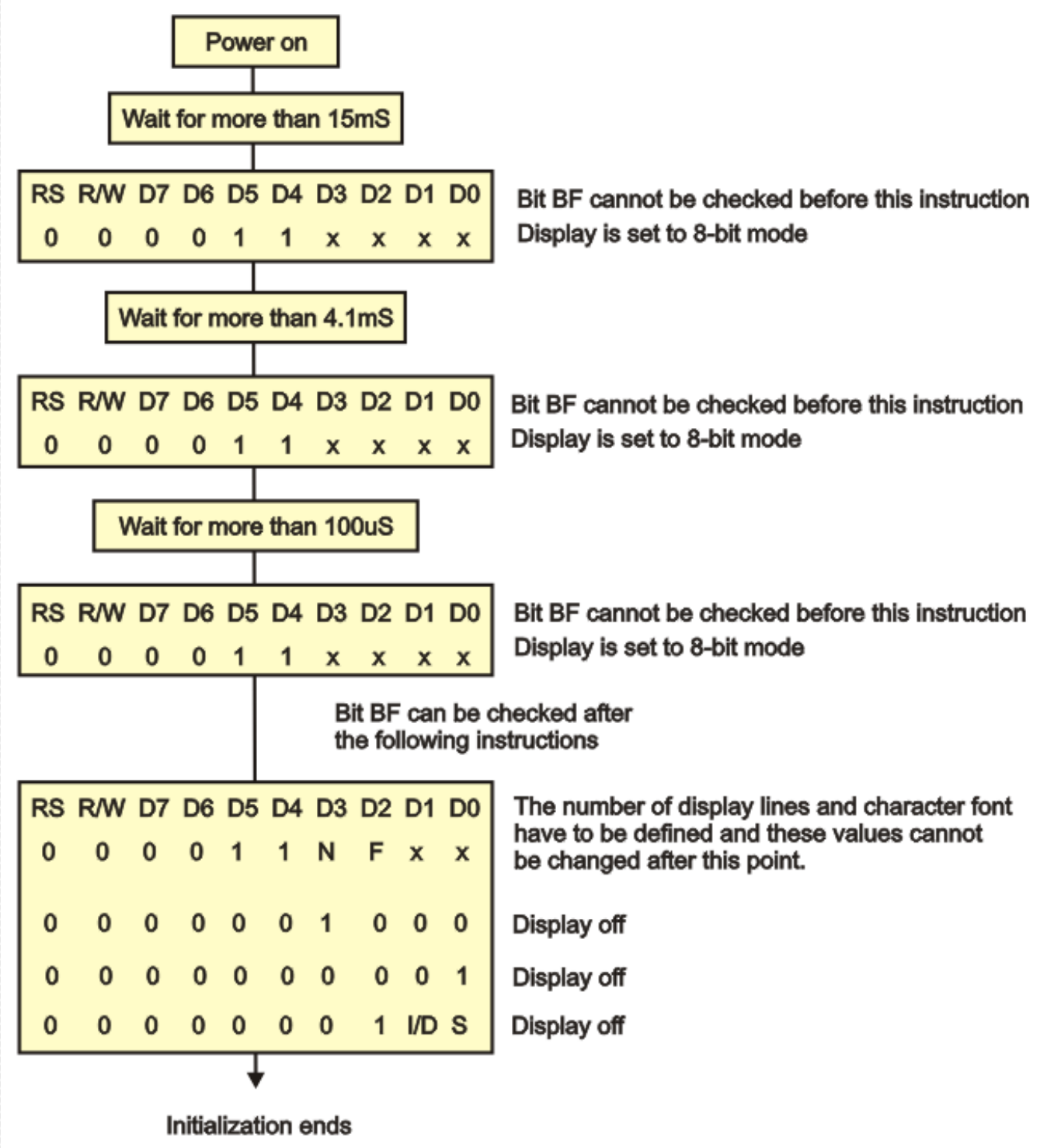
1. Display is cleared.
2. Mode
   DL = 1 Communication through 8-bit interface
   N = 0 Messages are displayed in one line
   F = 0 Character font 5 x 8 dots
3. Display/Cursor on/off
   D = 0 Display off
   U = 0 Cursor off
   B = 0 Cursor blink off

4. Character entry
   ID = 1 Displayed addresses are automatically incremented by 1
   S = 0 Display shift off

Automatic reset is mostly done without any problems. Mostly, but not always! If for any reason the power supply voltage does not reach full value within 10mS, the display will start performing completely unpredictably. If the voltage supply unit is not able to meet that condition or if it is needed to provide completely safe operation, the process of initialization is applied. Initialization, among other things, causes a new reset enabling display to operate normally.

Automatic reset is mostly done without any problems. Mostly, but not always! If for any reason power supply voltage does not reach full value within 10mS, display will start performing completely unpredictably. If voltage supply unit is not able to meet that condition or if it is needed to provide completely safe operation, the process of initialization is applied. Initialization, among other things, causes a new reset enabling display to operate normally.

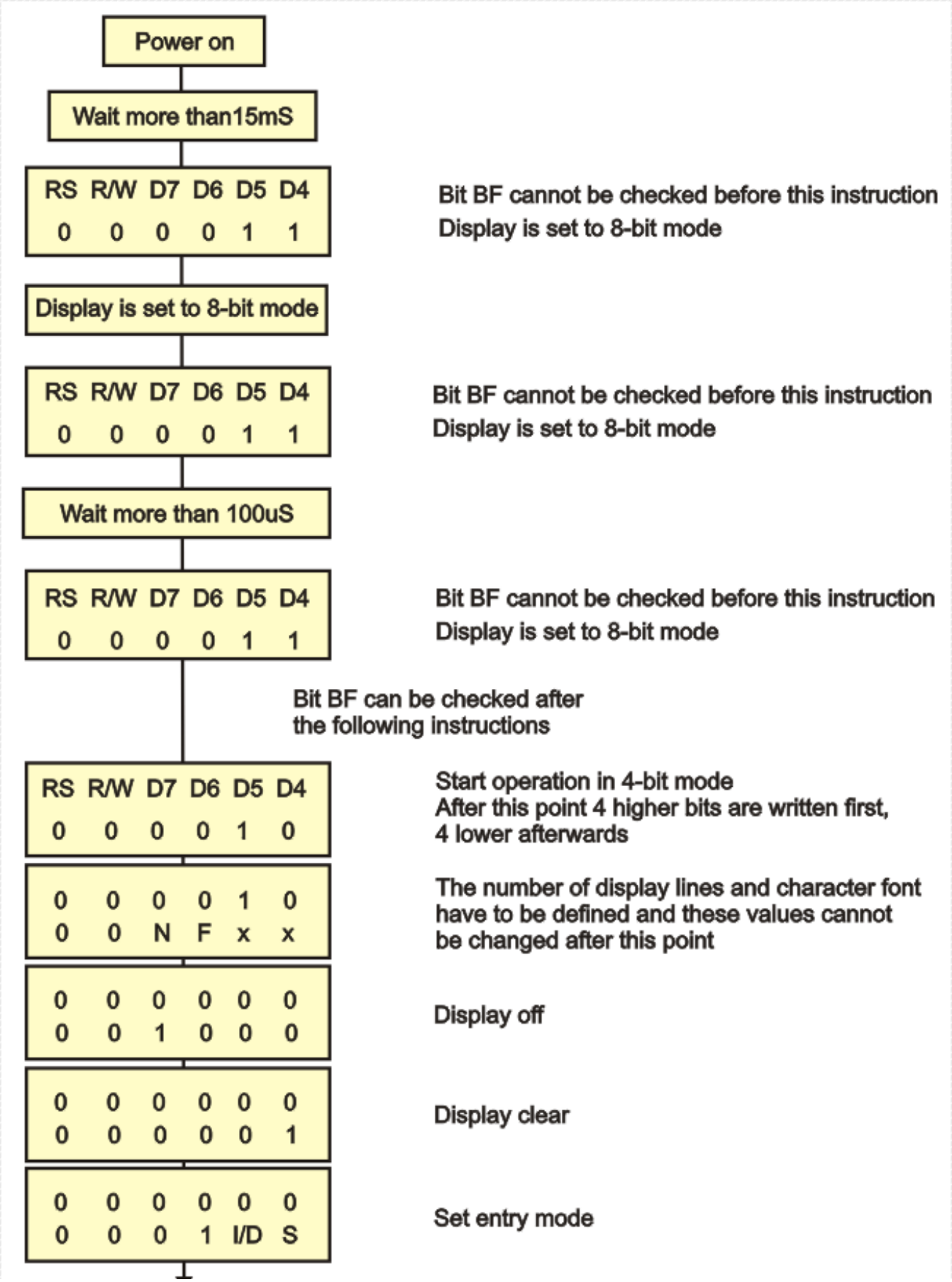
Refer to the figure below for the procedure on 8-bit initialization:

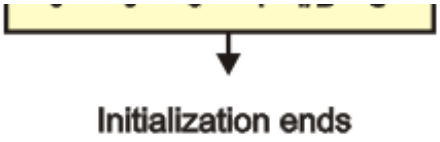


Initialization ends

It is not a mistake! In this algorithm, the same value is transferred three times in a row.

In case of 4-bit initialization, the procedure is as follows:

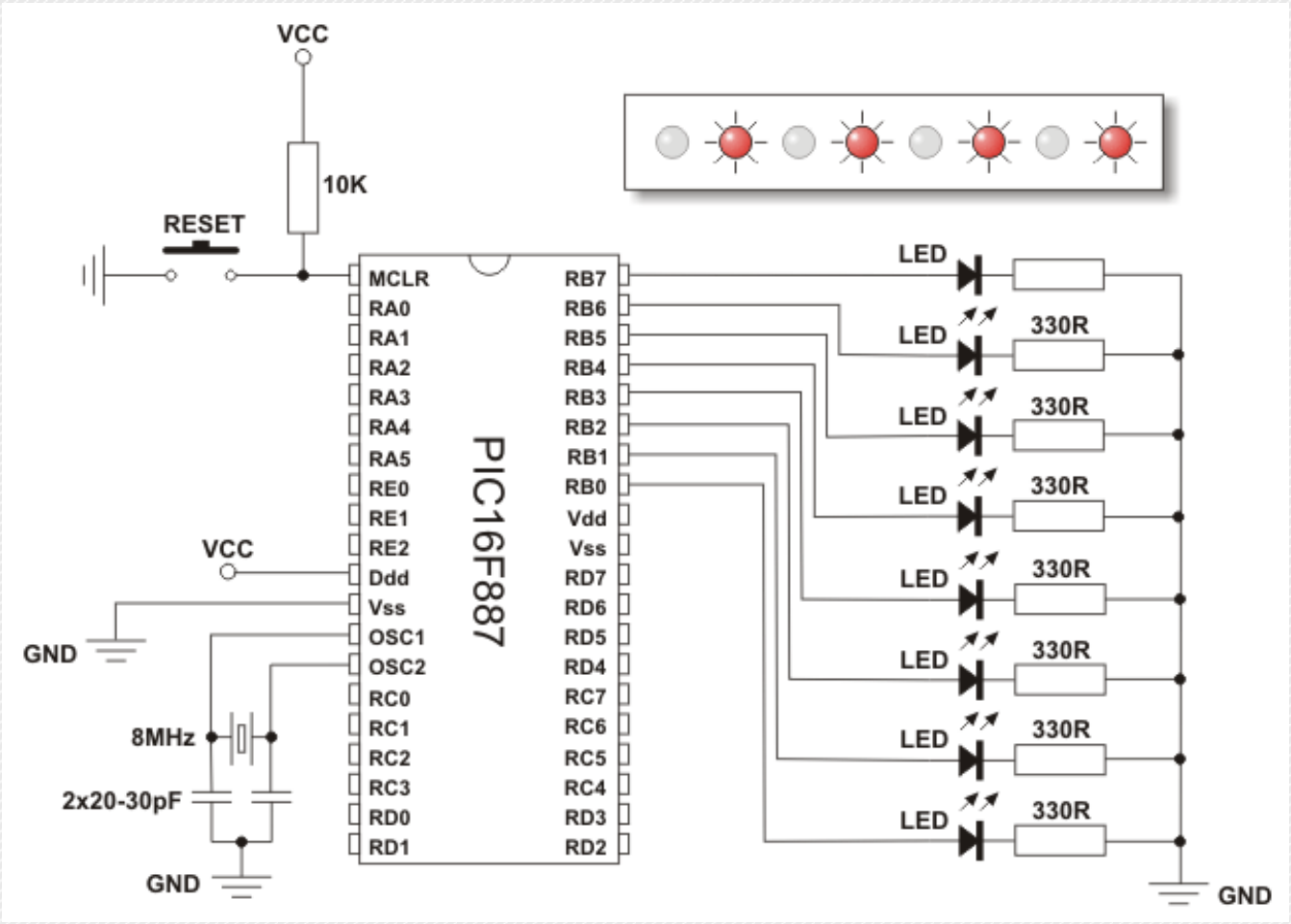




EXAMPLE 1

Writing header and configuring I/O pins

The only purpose of this program is to turn on a few LED diodes on port B. It is nothing special. Anyway, use this example to study what a real program looks like. The figure below shows a connection scheme, while the program is on the next page.



When switching on, every other LED diode on the port B emits light. That is enough to indicate that the microcontroller is properly connected and operates normally.

This example gives the description of a correctly written header and a few initial directives. They represent a part of the program used in all programs described in this book. To skip repetitiveness, it will not be written in the following examples, but is considered to be at the beginning of every program (marked as a "Header").

Example 1

```
;*****************************************************
;             Name:         Test.asm
;             Date:         November 19, 2007
;             Version:      1.00
;             Programmer:   James Jones
;*****************************************************************************
;             Description: Testing microcontroller
;*****************************************************************************

      list          p=16f887            ; Type of microcontroller
      #include      <p16f887.inc>       ; Defines all SFRs
                                        ; and bits within the PIC16F887

      errorlevel    -302   ; Disables message "Register
                           ; in operand not in bank 0. Ensure that
                           ; bank bits are correct."
;*****************************************************************************

      __CONFIG      _CONFIG1, _HS_OSC & _WDT_OFF & _PWRTE_ON & _MCLRE_ON
      & _CP_OFF & _CPD_OFF & _BOR_ON & _IESO_ON & _FCMEN_ON & _LVP_OFF
      & _DEBUG_OFF

      __CONFIG      _CONFIG2, _BOR40V & _WRT_OFF
;*****************************************************************************

      ORG           0x0000        ; Address of the first program
                                  ; instruction
                                  ; RESET vector)

      banksel       TRISB         ; Selects bank containing TRISB
      clrf          TRISB         ; All port B pins are configured
                                  ; as outputs
      banksel       PORTB         ; Selects bank containing PORTB
      movlw         B'01010101'   ; Moves number 01010101 to W
      movwf         PORTB         ; Moves number 01010101 from W to PORTB

      end                         ; End
```

Header

Program Instruction

Config word should be displayed in one line

The purpose of the header and initial directives is briefly described below.

Header:

The header is placed at the beginning of the program and gives basic information in the form of comments (name of the program, release date etc.). Don't be deluded into thinking that after a few months you will know what that program is about and why it is saved in your computer.

Initial directives:

```
list p=16f887
```

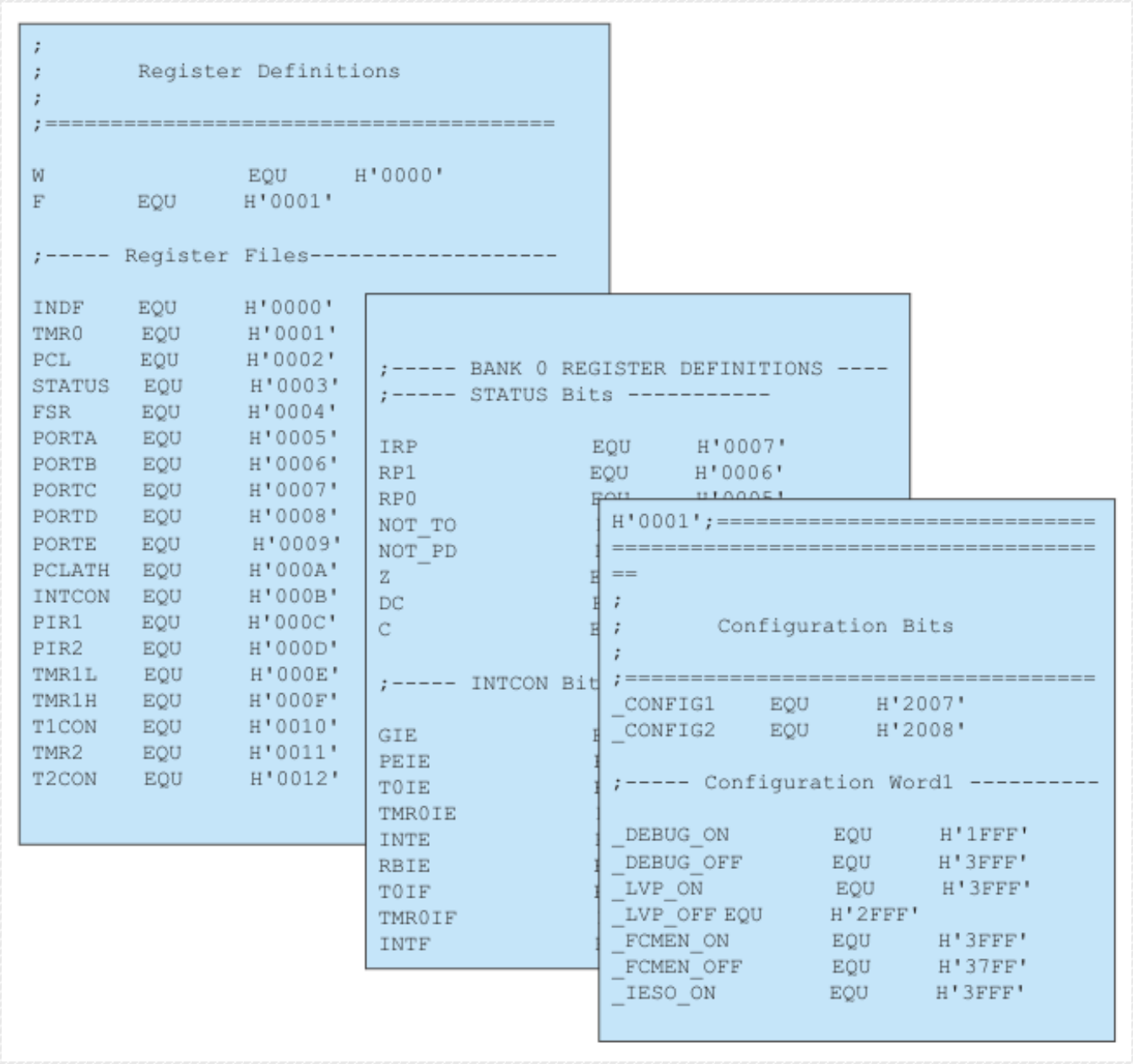
This directive defines processor to execute a program.

```
#include <p16f887.inc>
```

It enables the compiler to access the document p16f887.inc (If you have MPLAB installed, it is placed by default on C:\Program files\Microchip\MPASM Suite). Every SFR register contained in this document, as well as every bit, has its own name and address. If the program reads for example:

```
bsf INTCON, GIE
```

It means that the GIE bit of the INTCON register should be set. Instruction, as such, makes no sense to the compiler. It has to access the ".inc" document in order to know that the seventh bit of the SFR at the address 000B hex should be set.



errorlevel -302

This is a "cosmetic" directive which disables the irritating message "Register in operand not in ..." to appear at the end of every compiling process. It is not necessary, but useful.

__config

This directive is used to include config word in the program upon compiling. It is not necessary because the same operation is performed by software for loading program into chip. However, do you have any idea which software will be used by the end user? What options will be set by default? You are the end user?! Do you know which program you will be using for MCU programming next year? Make life easier for yourself, take this directive as a necessary one and include it in your program.

EXAMPLE 2

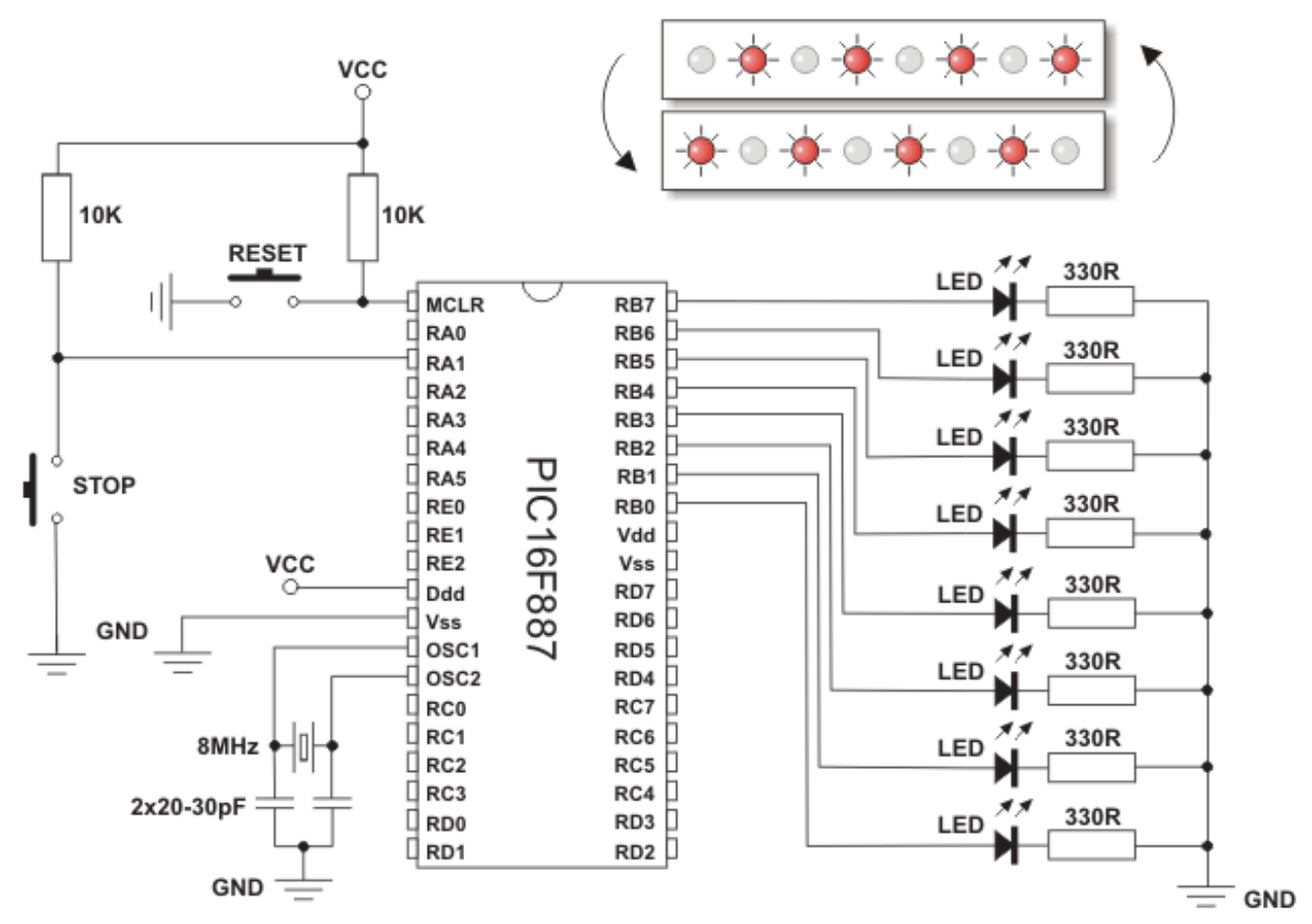
Using program loop and internal oscillator LFINTOSC

This is a continuation of the previous example, but deals with a bit more complicated problem... The idea is to make the LED diodes on the port B blink. A simple thing at first glance! It is enough to periodically change logic state on the port B. In this case, numbers 01010101 and 10101010 are selected to change in the following way:

1. Set binary combination 01010101 on port B;
2. Remain in loop1;
3. Replace existing bits combination on port B with 10101010;
4. Remain in loop2; and
5. Return to the step 1 and repeat the whole procedure.

Do you know how fast this should be done? It would be possible to observe changes on port B only if, apart from the delays provided in loop1 and loop2, the whole process is slowed down approximately 250 times more. Because of this, the microcontroller uses internal oscillator LFINTOSC with the frequency of 31kHz instead of the external oscillator with quartz crystal (8MHz).

You have noticed that the clock signal source has changed "on the fly". If you want to make sure of it, remove quartz crystal prior to switching the microcontroller on. What will happen? The microcontroller will not start operating because the config word loaded with the program requires the use of the crystal on switching on. If you remove the crystal later during the operation, it will not affect the microcontroller at all!



Example 2:

```
;*******************************************************************
;                   Header
;*******************************************************************
;************** DEFINING VARIABLES *********************************
       cblock  0x20            ; Block of variables starts at address 20h
       counter1                ; Variable "counter1" at address 20h
       endc
```

```
;*******************************************************************
       org     0x0000          ; Address of the first program instruction

       banksel OSCCON          ; Selects memory bank containing
                               ; register OSCCON
       bcf     OSCCON,6        ; Selects internal oscillator LFINTOSC with
       bcf     OSCCON,5        ; the frequency of 31KHz
       bcf     OSCCON,4
       bsf     OSCCON,0        ; Microcontroller uses internal oscillator

       banksel TRISB           ; Selects bank containing register TRISB
       clrf    TRISB           ; All port B pins are configured as outputs
       banksel PORTB           ; Selects bank containing register PORTB

loop
       movlw   B'01010101'     ; Binary number 01010101 is written to W
       movwf   PORTB           ; Number is moved to PORTB
       movlw   h'FF'           ; Number hFF is moved to W
       movwf   counter1        ; Number is moved to variable "counter1"

loop1
       decfsz  counter1        ; Variable "counter1" is decremented by 1
       goto    loop1           ; If result is 0, continue. If not,
                               ; remain in loop1

       movlw   B'10101010'     ; Binary number 10101010 is moved to W
       movwf   PORTB           ; Number is moved to PORTB
       movlw   h'FF'           ; Number hFF is moved to W
       movwf   counter1        ; Number is moved to variable "counter1"

loop2
       decfsz  counter1        ; Variable "counter1" is decremented by 1
       goto    loop2           ; If result is 0, continue. If not,
                               ; remain in loop2

       goto    loop            ; Go to label loop
       end                     ; End of program
```

EXAMPLE 3

Using nested loop

The connection scheme is again the same. To make this a bit more interesting, a different combination of port B bits change each other. And, that's not all of course. As seen from the previous two examples, the microcontroller is very fast and often, it needs to be slowed down. The use of the built-in oscillator LF, as in example 2, is the last measure that should be applied. The problem is more often solved by using nested loops in a program. In this example, the variable "counter1" is decremented 255 times by 1 in the shorter loop1. Prior to leaving this loop, the program will countdown 255 times from 255 to 0. It means that between only two LED diode's blink on the port, there are 255x255 pulses coming from the quartz oscillator. Precisely speaking, the number of pulses amounts to approximately 196 000 since it also takes some time to execute jump instructions and decrement instructions. Yes, it's true, the microcontroller mostly waits and does nothing...

Example 3:

```
;****************** Header ********************************************
;************* DEFINING VARIABLES ***************************************

       cblock      0x20          ; Block of variables starts at address 20h
       counter1                  ; Variable "counter1" at address 20h
       counter2                  ; Variable "counter2" at address 21h
       endc
;*******************************************************************
       org         0x0000        ; Address of the first program instruction

       banksel     TRISB         ; Selects bank containing register TRISB
       clrf        TRISB         ; Clears TRISB

       banksel     PORTB         ; Selects bank containing register PORTB
loop
```

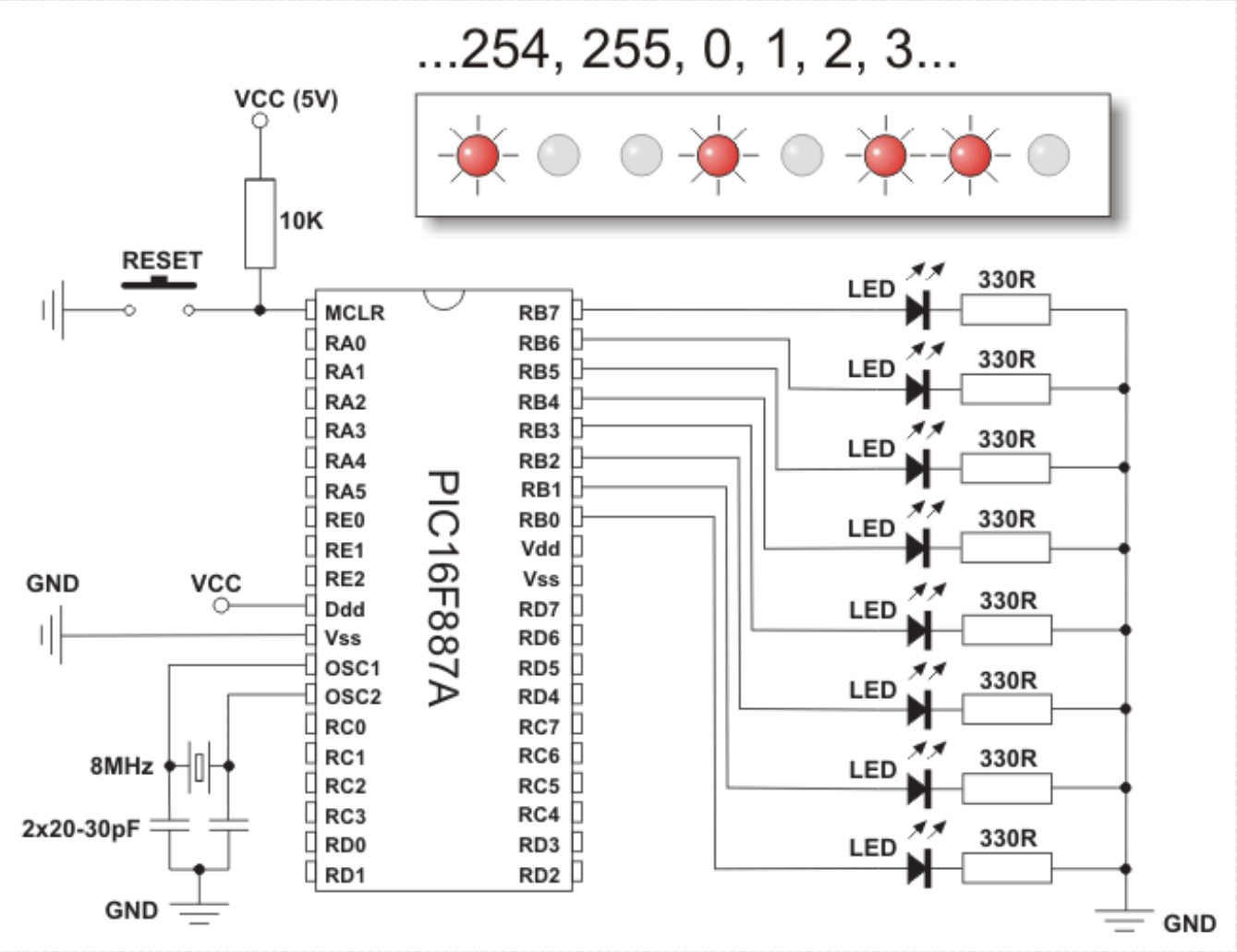
```
        movlw       B'11110000'   ; Binary number 11110000 is moved to W
        movwf       PORTB         ; Number is moved to PORTB
        movlw       h'FF'         ; Number hFF is moved to W
        movwf       counter2      ; Number is moved to variable "counter2"

loop2
        movlw       h'FF'         ; Number hFF is moved to W
        movwf       counter1      ; Number is moved to "counter1"
loop1
        decfsz      counter1      ; Decrements "counter1" by 1. If result is 0
        goto        loop1         ; skip next instruction

        decfsz      counter2      ; Decrements "counter2" by 1. If result is 0
        goto        loop2         ; skip next instruction

        movlw       B'00001111'   ; Binary number 00001111 is moved to W
        movwf       PORTB         ; Number is moved to PORTB
        movlw       h'FF'         ; Number hFF is moved to W
        movwf       counter2      ; Number is moved to variable "counter2"

loop4
        movlw       h'FF'         ; Number hFF is moved to W
        movwf       counter1      ; Number is moved to variable "counter1"
loop3
        decfsz      counter1      ; Decrements "counter1" by 1. If result is 0
                                  ; skip next instruction
        goto        loop3
        decfsz      counter2      ; Decrements "counter2" by 1. If result is 0
        goto        loop4         ; skip next instruction
        goto        loop          ; Jump to label loop
        end                       ; End of program
```

Example 4

Using timer TMR0 and Interrupts

If you have read the previous example, you would have noticed a disadvantage of providing delays using loops. In all these cases, the microcontroller is "captive" and does nothing. It simply waits for some time to pass. Such wasting of time is an unacceptable luxury and some other method should be applied.

Do you remember the story about the timers? About interrupts? This example makes links between them in a practical way. The schematic is still the same as well as the challenge. It is necessary to provide delay long enough to notice changes on a port. This time, the timer TMR0 with the assigned prescaler is used for that purpose. Interrupt occurs on every timer register overflow and interrupt routine increments the number in port B by 1. The whole procedure is performed "behind the scenes" of the whole process, which enables the microcontroller to do other things.



Pay attention to a few details:

- Even though it is unnecessary in this case, the contents of the most important registers (W, STATUS and PCLATH) must be saved at the beginning of the interrupt routine;
- Interrupt causes the appropriate flag bit to be automatically set and the GIE bit to be automatically cleared. At the end of the interrupt routine, do not forget to return these bits to the state they had prior to the interrupt occurring; and
- At the end of the interrupt rutine, important registers should be given the original content.

Example 4:

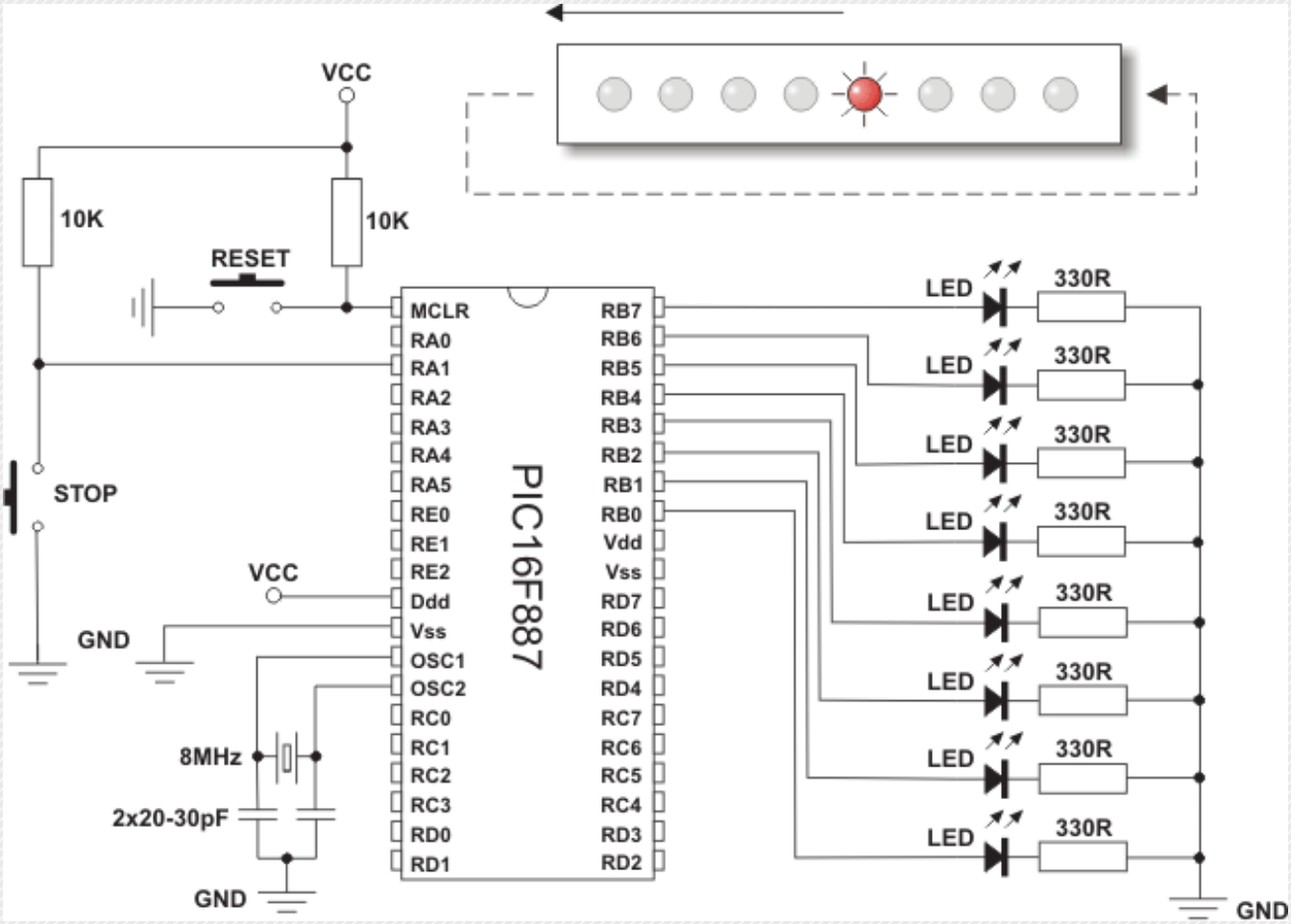
```
;********************** Header ********************************************
;***************** DEFINING VARIABLES *****************************************

       cblock      0x20            ; Block of variables starts at address 20h
       w_temp                      ; Variable at address 20h
       pclath_temp                 ; Variable at address 21h
       status_temp                 ; Variable at address 22h
       endc

;************************ START OF PROGRAM ********************************
       org         0x0000          ; Address of the first program instruction
       goto        main            ; Go to label "main"

;************************ INTERRUPT ROUTINE ********************************
       org         0x0004          ; Interrupt vector
       movwf       w_temp          ; Saves value in register W
       movf        STATUS          ; Saves value in register STATUS
       movwf       status_temp
       movf        PCLATH          ; Saves value in register PCLATH
```

```
        movwf       pclath_temp

        banksel     PORTB           ; Selects bank containing PORTB
        incf        PORTB           ; Increments register PORTB by 1

        banksel     INTCON          ; Selects bank containing INTCON
        bcf         INTCON,TMR0IF   ; Clears interrupt flag TMR0IF

        movf        pclath_temp,w   ; PCLATH is given its original content
        movwf       PCLATH
        movf        status_temp,w   ; STATUS is given its original content
        movwf       STATUS
        swapf       w_temp,f        ; W is given its original content
        swapf       w_temp,w

        bsf         INTCON,GIE      ; Global interrupt enabled
        retfie                      ; Return from interrupt routine

;********************** MAIN PROGRAM ************************************
main                                ; Start of the main program

        banksel     ANSEL           ; Bank containing register ANSEL
        clrf        ANSEL           ; Clears registers ANSEL and ANSELH
        clrf        ANSELH          ; All pins are digital

        banksel     TRISB           ; Selects bank containing register TRISB
        clrf        TRISB           ; All port B pins are configured as outputs

        banksel     OPTION_REG      ; Bank containing register OPTION_REG
        bcf         OPTION_REG,T0CS ; TMR0 counts pulses from oscillator
        bcf         OPTION_REG,PSA  ; Prescaler is assign to timer TMR0

        bsf         OPTION_REG,PS0  ; Prescaler rate is 1:256
        bsf         OPTION_REG,PS1
        bsf         OPTION_REG,PS2

        banksel     INTCON          ; Bank containing register INTCON
        bsf         INTCON,TMR0IE   ; TMR0 interrupt overflow enabled
        bsf         INTCON,GIE      ; Global interrupt enabled

        banksel     PORTB           ; Bank containing register PORTB
        clrf        PORTB           ; Clears port B
loop
        goto        loop            ; Remain here
        end                         ; End of program
```

Example 5

Using subroutine, using push-buttons

In the previous examples the microcontroller executes the program without being influenced in any way its surrounding. In practice, devices operating in this way are very rare (for example, simple neon signs). You guess, among other components, input pins will also be used in this example. There is a schematic in the figure below, while the program is on the next page. Everything is still very simple.



At the beginning of the program, immediately upon defining variables, the microcontroller pins* are configured by using registers TRISA and TRISB.

In the main program, one bit on port B is set first. Then the contents of this register is constantly moved by one place to the left (instruction rlf PORTB). It gives us the impression that the lit LED diodes is moving. To make it visible, the whole process must be slow enough. Press on the push-button "STOP" stops the movement and the program remains in loop3. Delay is provided by means of a nested loop. This time, it is placed in a short subroutine "DELAY".

** It is not necessary for PORTA pins since they are automatically configured as inputs after every reset.*

Example 5:

```
;*******************************************************************************
;                       Header
;*******************************************************************************
;************** DEFINING VARIABLES ********************************************

        cblock      0x20            ; Block of variables starts at address 20h
        counter1                    ; Variable "counter1" at address 20h
        counter2                    ; Variable "counter2" at address 21h
        endc                        ; Block of variables ends

;************************ MAIN PROGRAM ************************************

        org         0x0000          ; Address of the first program instruction
        banksel     ANSEL           ; Selects bank containing register ANSEL
        clrf        ANSEL           ; Clears registers ANSEL and ANSELH to
        clrf        ANSELH          ; configure all inputs as digital

        banksel     TRISB           ; Selects bank containing register TRISB
        clrf        TRISB           ; All port B pins are configured as outputs
        movlw       B'00000010'
```

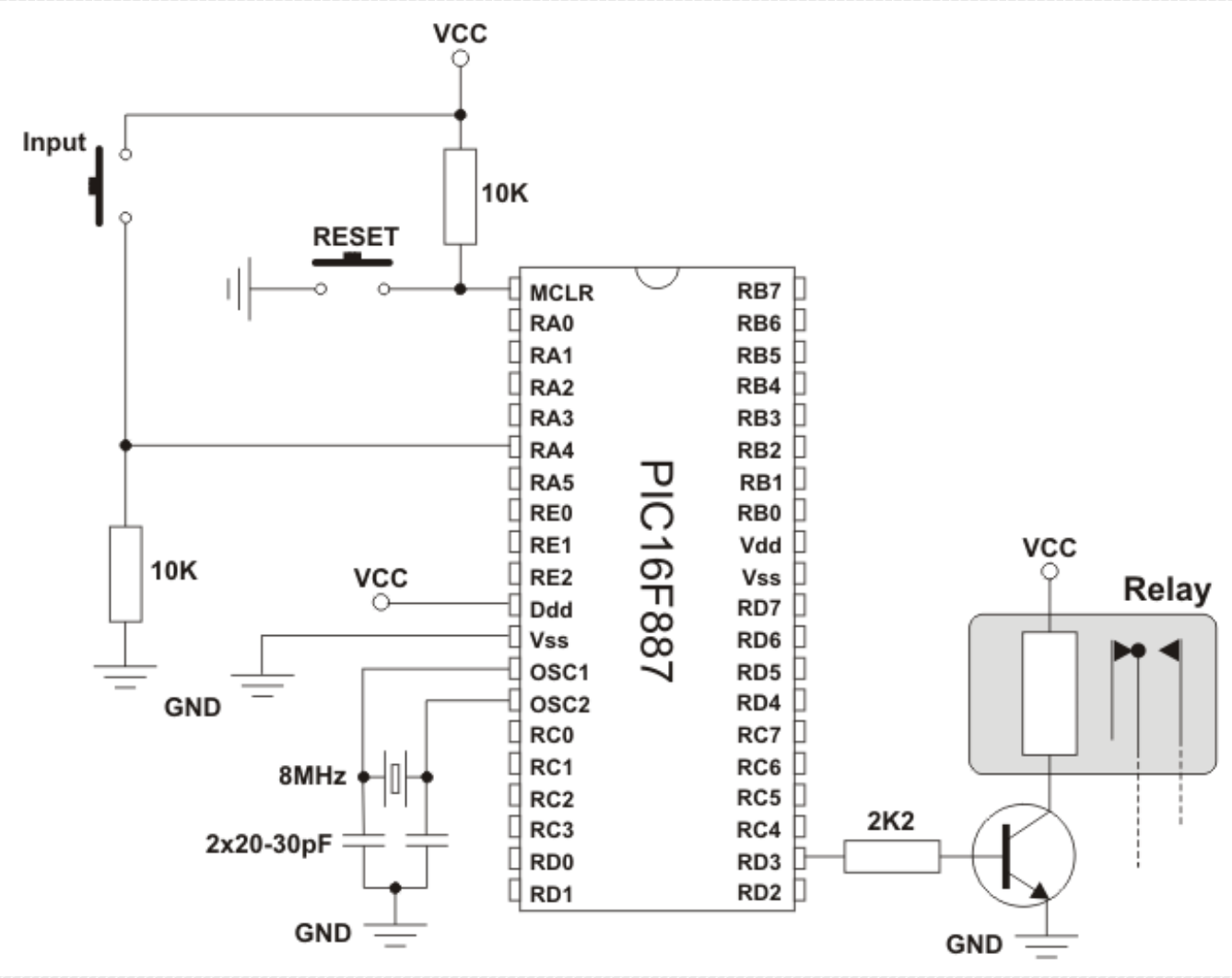
```
        movwf       TRISA           ; Pin RA1 is input

        banksel     PORTB           ; Selects bank containing register TRISB
        movlw       B'00000001'     ; Writes 1 to register W
        movwf       PORTB           ; Number is moved to PORTB
loop
        rlf         PORTB           ; Port B bits rotates by one place left
        call        DELAY           ; Calls subroutine "DELAY"
loop3
        btfss       PORTA,1         ; Tests the firs port A bit
        goto        loop3           ; "0" is applied to pin.Go to label "loop3"
        goto        loop            ; "1" is applied to pin.Go to label "loop"

;*********************** SUBROUTINES ************************************
DELAY
        clrf        counter2        ; Clears variable "counter2"
loop1
        clrf        counter1        ; Clears variable "counter1"
loop2
        decfsz      counter1        ; Decrements variable "counter1" by 1
        goto        loop2           ; Result is not 0. Go to label loop2
        decfsz      counter2        ; Decrements variable "counter2" by 1
        goto        loop1           ; Result is not 0. Go to lab loop1
        return                      ; Return from subroutine "DELAY"

        end                         ; End of program
```

EXAMPLE 6

TMR0 as a counter, defining new variables, using relay

This time, TMR0 is used as a counter. The idea is to connect the counter input to one pushbutton so that it counts one pulse at a time upon every button press. When the number of counted pulses becomes equal to the number in register TEST, logic one voltage (5V) will be applied to the PORTD, 3 pin. Since this voltage activates an electro-mechanical relay, this bit is called the same- "Relay".

In this example, the TEST register contains number 5. Naturally, it could be any number and could be calculated or entered via the keyboard. Instead of a relay, the microcontroller can activate some other device and instead of push-buttons it can use sensors. This example illustrates one of the most common uses of the microcontroller in industry. When something is done as many times as needed, then something else should be switched on or off...



Example 6:

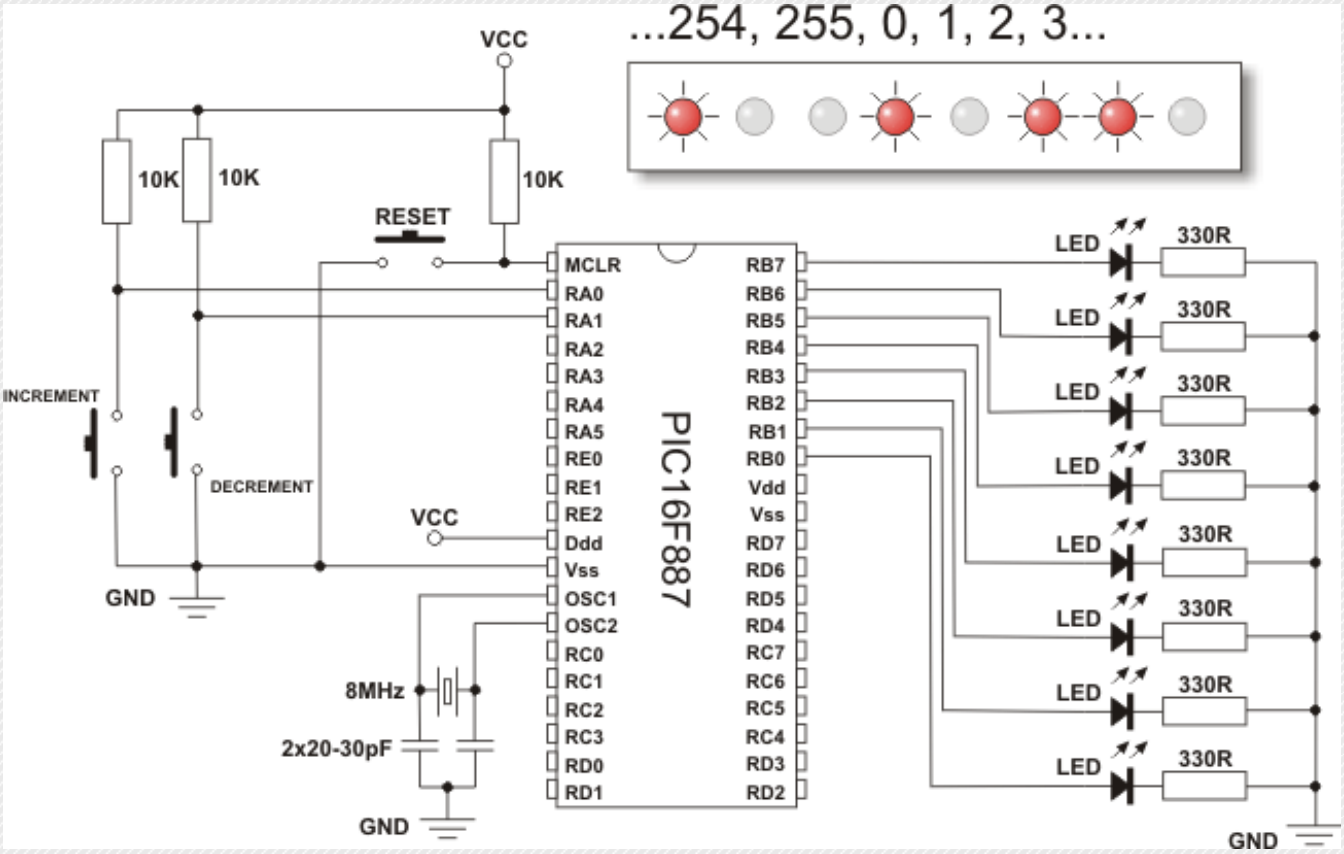
```
;*******************************************************************************
;                       Header
;*******************************************************************************
;************** DEFINING VARIABLES ********************************************

       TEST        equ B'00000101' ; Binary number 00000101 = TEST
       #define     RELAY PORTD,3   ; Pin PORTD,3 = RELAY

;************************ MAIN PROGRAM ***********************************

       org         0x0000          ; Address of the first program instruction

       banksel     TRISB           ; Selects bank containing register TRISB
       clrf        TRISB           ; All port B pins are configured as outputs
       clrf        TRISD           ; All port D pins are configured as outputs
       movlw       B'00010000'     ; This number is written to W register
       movwf       TRISA           ; Only the forth pin of port A is input

       banksel     OPTION_REG      ; Bank containing OPTION_REG register
       bsf         OPTION_REG,T0CS ; Pin RA4 is supplied with pulses
       bsf         OPTION_REG,PSA  ; Prescaler rate is 1:1

       banksel     PORTB           ; Selects bank containing PORTB register

       clrf        TMR0            ; Clears timer register
       bcf         PORTD,3         ; Pin PORTD,3 = 0
loop
       movfw       TMR0            ; Timer register is moved to W register
       movwf       PORTB           ; W register is moved to PORTB
       xorlw       TEST            ; Operation exclusive OR between
```

```
                                ; W register and number TEST (00000101)
        btfsc       STATUS,Z    ; If numbers are equal, result is 0 and
        bsf         PORTD,3     ; bit STATUS,Z = 1. Bit PORTD,3 is set
        goto        loop        ; and jump to label loop is executed

        end                     ; End of program
```

EXAMPLE 7

Using macros in the program, using debounce routine

You have probably noticed in the previous example that the microcontroller does not always operate as expected. Namely, by pressing the push-button, the number on port B is not always incremented by 1. Mechanical push-buttons make several short successive contacts when they have been activated. You guess, the microcontroller registers and counts all that...



There are several ways to solve this problem. This program uses program delay known as *debounce*. Basically, it is a simple procedure. Upon input change detection (button press), a short program delay is provided and the program waits for another change (button release). Only after this, the program comes to a conclusion that the button is activated.

In this very case, the push-button is tested by means of macro called *button*. Besides, this macro contains a program delay which is provided by means of another macro *pausems*.

The main program is relatively simple and enables the variable "cnt" to be incremented and decremented by using two push-buttons. This variable is thereafter copied to port B and affects the LED (logic one (1) turns LED diode on, while logic zero (0) turns LED diode off).

Example 7:

```
;********************** Header **********************
;************* DEFINING VARIABLES ************************

        cblock      0x20            ; Block of variables starts at address 20hex
```

```
        HIcnt
        LOcnt
        LOOPcnt
        cnt
        endc                            ; End of block of variables
;*****************************************************************
        ORG         0x000               ; Reset vector
        nop
        goto        main                ; Go to program start (label "main")
;*****************************************************************

        include    "pause.inc"
        include    "button.inc"

;*****************************************************************
main
        banksel      ANSEL              ; Selects bank containing ANSEL
        clrf         ANSEL              ; All pins are digital
        clrf         ANSELH

        banksel      TRISB
        bsf          TRISA, 0
        bsf          TRISA, 1
        clrf         TRISB

        banksel      PORTB
        clrf         cnt
Loop
        button       PORT,0,0,Increment
        button       PORT,1,0,Decrement
        goto         Loop

Increment
        incf         cnt, f
        movf         cnt, w
        movwf        PORTB
        goto         Loop
Decrement
        decf         cnt, f
        movf         cnt, w
        movwf        PORTB
        goto         Loop
        end                             ; End of program
```

Macro "pausems"

```
;*****************************************************************
pausems MACRO arg1
        local        Loop1
        local        dechi
        local        Delay1ms
        local        Loop2
        local        End

        movlw        High(arg1)         ; Higher byte of argument is moved
                                        ; to HIcnt
        movwf        HIcnt
        movlw        Low(arg1)          ; Lower byte of argument is moved
                                        ; to LOcnt
        movwf        LOcnt
Loop1
        movf         LOcnt, f           ; Decrements HIcnt and LOcnt while
        btfsc        STATUS, Z          ; needed and calls subroutine Delay1ms

        goto         dechi
        call         Delay1ms
        decf         LOcnt, f
        goto         Loop1
dechi
        movf         HIcnt, f
```

```
        btfsc         STATUS, Z
        goto          End
        call          Delay1ms
        decf          HIcnt, f
        decf          LOcnt, f
        goto          Loop1
Delay1ms:                       ; Delay1ms provides delay of
        movlw         .100      ; 100*10us=1ms
        movwf         LOOPcnt   ; LOOPcnt<-100
Loop2:
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        decfsz      LOOPcnt, f
        goto        Loop2       ; Execution time of Loop2
        return                  ; is 10 us
End
        ENDM
;*****************************************************************
```

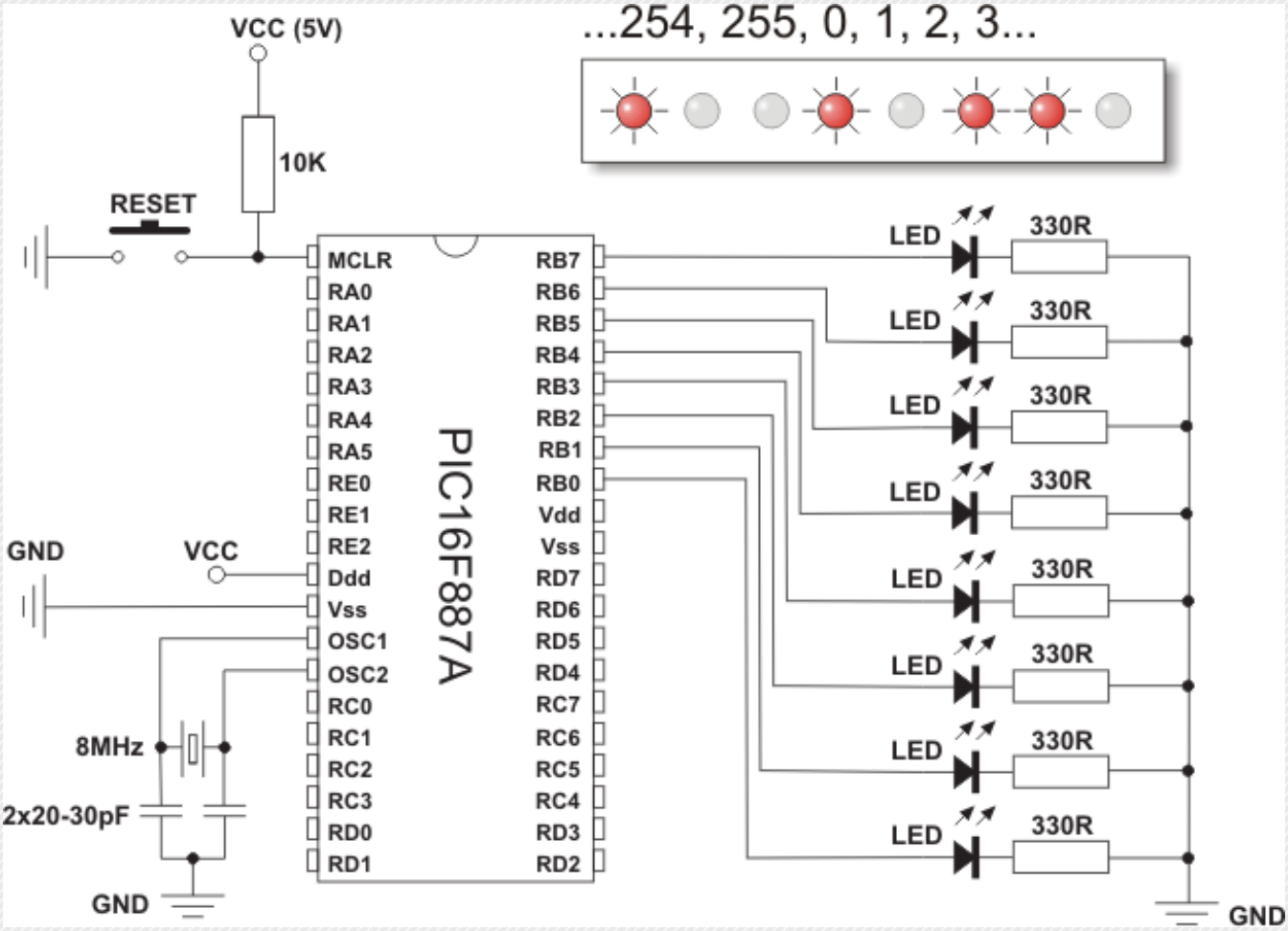
Macro "button"

```
;************************************************
;
button MACRO port,pin,hilo,label
        local       Pressed1        ; All labels are local
        local       Pressed2
        local       Exit1
        local       Exit2

        IFNDEF      debouncedelay   ; Enables debounce time to be defined
                                    ; in main program
        #define     debouncedelay .10
        ENDIF

        IF (hilo == 0)              ; If pull-up used
        btfsc       port, pin       ; If "1", push-button is pressed
        goto        Exit1
        pausems     debouncedelay   ; Wait for 10ms debounce
Pressed1
        btfss       port, pin
        goto        Pressed1
        pausems     debouncedelay   ; Wait until released and
        goto        label           ; jump to specified address
Exit1
        ELSE                        ; If pull-down used
        btfss       port, pin
        goto        Exit2           ; If "0", push-button is released
        pausems     debouncedelay   ; Wait for 10ms debounce
Pressed2
        btfsc       port, pin
        goto        Pressed2
        pausems     debouncedelay   ; Wait until released and
        goto        label           ; jump to specified address
Exit2
        ENDIF

        ENDM
;*****************************************************************
```

EXAMPLE 8

Using timer TMR1 and using interrupt

16-bit timer TMR1 is used in this example. By occupying its registers TMR1L and TMR1H, an interrupt occurs and the number on port B is incremented. This has already been seen in the previous examples. The difference is in the program delay which is a bit longer this time because the prescaler rate is 1:8.



Example 8:

```
;************************** Header ****************************************
;************* DEFINING VARIABLES *****************************************

       cblock      0x20            ; Block of variables starts at address 20h
       w_temp                      ; Variable at address 20h
       pclath_temp                 ; Variable at address 21h
       status_temp                 ; Variable at address 22h
       endc

;************************ PROGRAM START ***********************
       org         0x0000          ; Address of the first program instruction
       goto        main            ; Jump to label "main"

;************************ INTERRUPT ROUTINE *********************************

       org         0x0004          ; Interrupt vector
       movwf       w_temp          ; Save register W

       movf        STATUS          ; Save register STATUS
       movwf       status_temp

       movf        PCLATH          ; Save register PCLATH
       movwf       pclath_temp

       banksel     PORTB           ; Selects bank containing PORTB
       incf        PORTB           ; Register PORTB is incremented by 1

       movf        pclath_temp,w   ; PCLATH is given its original content
       movwf       PCLATH
```

```
        movf        status_temp,w   ; STATUS is given its original content
        movwf       STATUS
        swapf       w_temp,f        ; W is given its original content
        swapf       w_temp,w

        banksel     PIR1            ; Selects bank containing PIR1
        bcf         PIR1,TMR1IF     ; Clears interrupt flag TMR1IF

        bsf         INTCON,GIE      ; Global interrupt enabled
        retfie                      ; Return from interrupt routine

;*********************** MAIN PROGRAM ************************************

main                                ; Start of main program
        banksel     ANSEL           ; Selects bank containing register ANSEL
        clrf        ANSEL           ; Clears registers ANSEL and ANSELH
        clrf        ANSELH          ; All pins are digital

        banksel     TRISB           ; Selects bank containing register TRISB
        clrf        TRISB           ; All port B pins are configured as outputs

        banksel     T1CON           ; Selects bank containing register T1CON
        bcf         T1CON,TMR1CS    ; TMR1 counts pulses generated by oscillator

        bsf         T1CON,T1CKPS0   ; Prescaler rate is 1:8
        bsf         T1CON,T1CKPS1
        bsf         T1CON,TMR1ON    ; Turns on timer TMR1

        banksel     PIE1            ; Selects bank containing register PIE1
        bsf         PIE1,TMR1IE     ; TMR1 interrupt overflow enabled
        bsf         INTCON,PEIE     ; Peripheral modules interrupt enabled
                                    ; Timer TMR1 belongs to peripheral modules
        bsf         INTCON,GIE      ; Global interrupt enabled

        banksel     PORTB           ; Selects bank containing register PORTB
        clrf        PORTB           ; Clears port B
loop
        goto        loop            ; Remain here
        end                         ; End of program
```

EXAMPLE 9

Using timer TMR2, configuring quartz oscillator

This example illustrates the use of timer TMR2. The microcontroller uses internal oscillator HFINTOSC with the frequency of 500 kHz. The whole program works as follows: After the period of time defined by register PR, prescaler and postscaler has expired, an interrupt occurs. Interrupt routine decrements the content of the PR register and simultaneously increments the content of port B. Since the number in register PR, which determines when interrupt is to occur is constantly decremented, interrupt will occur for shorter and shorter periods of time. In other words, counting will be carried out faster. A new cycle of accelerated counting starts after every register PR overflow.

Example 9:

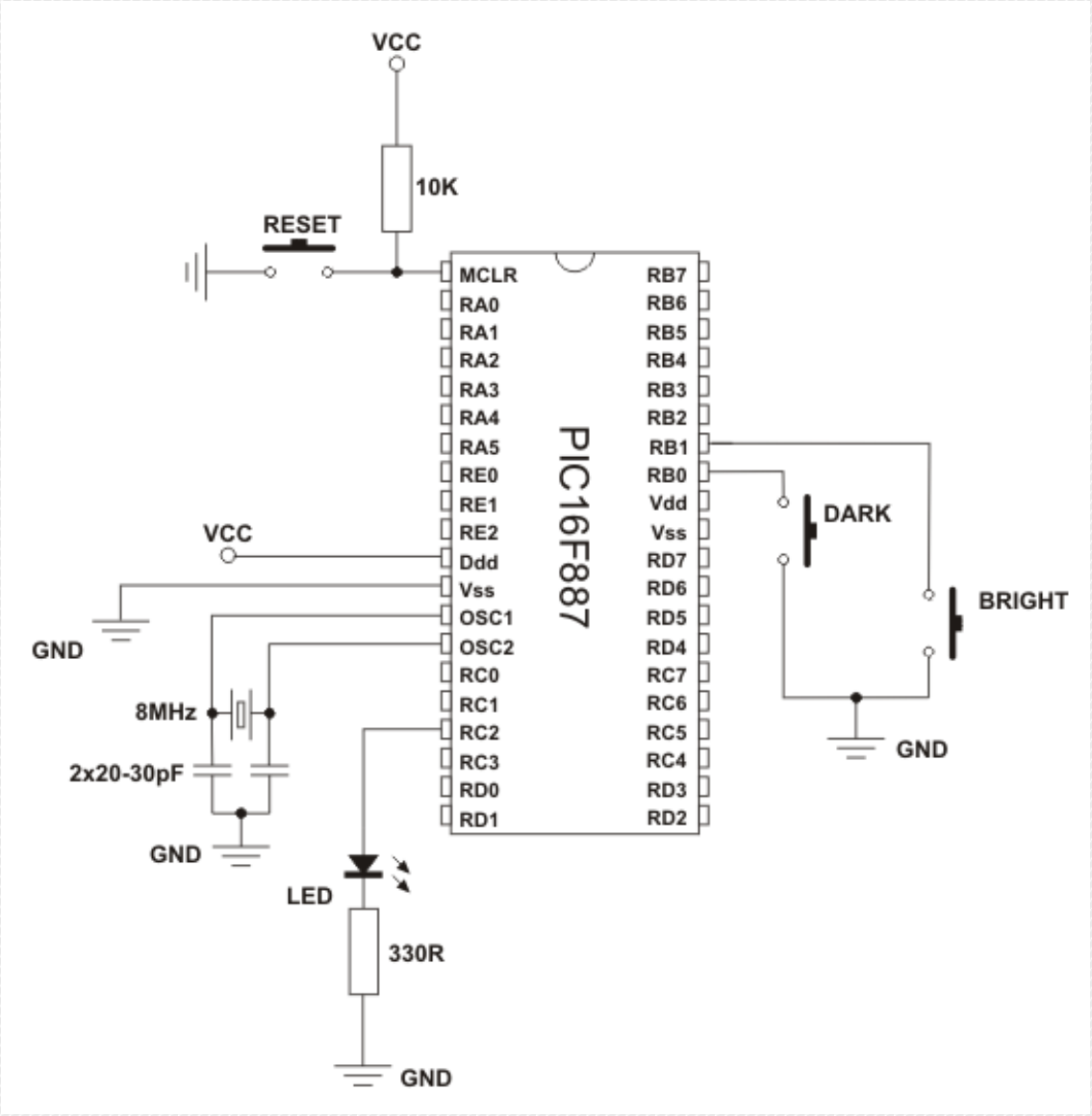
```
;********************** Header ************************************************
;************* DEFINING VARIABLES *********************************************

        cblock      0x20            ; Block of variables starts at address 20h
        w_temp                      ; Variable at address 20h
        pclath_temp                 ; Variable at address 21h
        status_temp                 ; Variable at address 22h
        endc

;*********************** PROGRAM START ***************************************
        org         0x0000          ; Address of the first program instruction
        goto        main            ; Jump to label "main"

;*********************** INTERRUPT ROUTINE ***********************************

        org         0x0004          ; Interrupt vector
```

```
        movwf       w_temp          ; Save register W

        movf        STATUS          ; Save register STATUS
        movwf       status_temp

        movf        PCLATH          ; Save register PCLATH
        movwf       pclath_temp

        banksel     PORTB           ; Selects bank containing PORTB
        incf        PORTB           ; Increments PORTB register by 1
        banksel     PR2             ; Selects bank containing PR2
        decf        PR2             ; PR2 is decremented by 1
        movf        pclath_temp,w   ; PCLATH is given its original state
        movwf       PCLATH
        movf        status_temp,w   ; STATUS is given its original state
        movwf       STATUS
        swapf       w_temp,f        ; W is given its original state
        swapf       w_temp,w

        banksel     PIR1            ; Selects bank containing PIR1
        bcf         PIR1,TMR2IF     ; Clears interrupt flag TMR2IF

        bsf         INTCON,GIE      ; Global interrupt enabled
        retfie                      ; Return from interrupt routine

;********************** MAIN PROGRAM ***************************************
main                                ; Start of the main program
        banksel     OSCCON          ; Selects bank containing register OSCCON
        bcf         OSCCON,6        ; Selects internal oscillator HFINTOSC with
        bsf         OSCCON,5        ; frequency of 500KHz
        bsf         OSCCON,4
        bsf         OSCCON,0        ; Microcontroller uses internal oscillator

        banksel     ANSEL           ; Selects bank containing register ANSEL
        clrf        ANSEL           ; Clears registers ANSEL and ANSELH
        clrf        ANSELH          ; All pins are digital

        banksel     TRISB           ; Selects bank containing register TRISB
        clrf        TRISB           ; All port B pins are configured as outputs
        clrf        PR2

        banksel     T2CON           ; Selects bank containing register T2CON
        movlw       H'FF'           ; Sets all control register bits
        movwf       T2CON           ; prescaler=1:16, postscaler=1:16 TMR2=ON
        clrf        PORTB

        banksel     PIE1            ; Selects bank containing register PIE1
        bsf         PIE1,TMR2IE     ; TMR2 interrupt enabled
        bsf         NTCON,PEIE      ; Peripheral modules interrupt enabled
                                    ; Timer TMR2 belongs to peripheral modules
        bsf         INTCON,GIE      ; Global interrupt enabled
loop
        goto        loop            ; Remain here
        end                         ; End of program
```

EXAMPLE 10

Module CCP1 as PWM signal generator

Since the CCP modules have a wide range of possibilities they are commonly used in practice. This example illustrates the use of CCP1 module in PWM mode. Bits of the CCP1CON register determine that the module operates as a single-output PWM. The same bits determine the PWM frequency to be 4.88 kHz. To make things more interesting, the duration of the output P1A (PORTC,2) pulses may be changed by means of push-buttons symbolically called "DARK" and "BRIGHT". Push-buttons are tested in interrupt routine initiated by the timer TMR1. Any change affects the LED diode so that it changes light intensity. Note that port B does not use external resistors because internal pull-up resistors are enabled. The whole process of generating PWM signal is performed "behind the scenes", which enables the microcontroller to do other things.



Example 10:

```
;********************** Header ********************************************
;************** DEFINING VARIABLES ****************************************
      cblock       0x20              ; Block of variables starts at address 20h
      w_temp                         ; Variable at address 20h
      pclath_temp                    ; Variable at address 21h
      status_temp                    ; Variable at address 22h
      endc

      #define      DARK PORTB,0      ; Push-button "DARK" is connected
                                     ; to PORTB,0 pin
      #define      BRIGHT PORTB,1    ; Push-button "BRIGHT" is connected
                                     ; to PORTB,1 pin
;************************ PROGRAM START ***********************************

      org          0x0000            ; First program instruction address
      goto         main              ; Jump to label "main"

;************************ INTERRUPT ROUTINE *******************************

      org          0x0004            ; Interrupt vector

      movwf        w_temp            ; Save register W

      movf         STATUS            ; Save register STATUS
      movwf        status_temp
```

```
      movf        PCLATH          ; Save register PCLATH
      movwf       pclath_temp

      banksel     CCPR1L
      btfss       DARK            ; Tests push-button "DARK"
      decf        CCPR1L          ; Push-button is pressed - decrement CCP1L by 1
      btfss       BRIGHT          ; Testing push-button "BRIGHT"
      incf        CCPR1L          ; Push-button is pressed - increment CCP1L by 1

      movf        pclath_temp,w   ; PCLATH is given its original content
      movwf       PCLATH
      movf        status_temp,w   ; STATUS is given its original content
      movwf       STATUS
      swapf       w_temp,f        ; W is given its original content
      swapf       w_temp,w

      banksel     PIR1            ; Selects bank containing PIR1
      bcf         PIR1,TMR1IF     ; Clears interrupt flag TMR1IF

      bsf         TMR1H,7         ; Accelerates timer TMR0 counting
      bsf         TMR1H,6         ;
      bsf         INTCON,GIE      ; Global interrupt enabled
      retfie                      ; Return from interrupt routine

;********************** MAIN PROGRAM ************************************

main                              ; Start of the main program
      banksel     ANSEL           ; Selects bank containing register ANSEL
      clrf        ANSEL           ; Clears registers ANSEL and ANSELH
      clrf        ANSELH          ; All pins are digital

      banksel     OPTION_REG      ; Selects bank containing register ANSEL
      bcf         OPTION_REG,7    ; Pull-up resistors enabled
      bsf         WPUB,0          ; Pull-up resistors enabled
      bsf         WPUB,1          ; on port B pins 0 and 1

      banksel     TRISC           ; Selects bank containing register TRISC
      clrf        TRISC           ; All port C pins are configured as outputs

      banksel      T1CON          ; Selects bank containing register T1CON
      bcf          T1CON,TMR1CS   ; TMR1 operates as a timer
      bcf          T1CON,T1CKPS0  ; Prescaler rate is 1:8
      bcf          T1CON,T1CKPS1
      bsf          T1CON,TMR1ON   ; Activates timer TMR1

      banksel      PIE1           ; Selects bank containing register PIE1
      bsf          PIE1,TMR1IE    ; Interrupt TMR1 is enabled
      bsf          INTCON,PEIE    ; Peripheral modules interrupts are
                                  ; enabled
      bsf          INTCON,GIE     ; Global interrupt enabled

      movlw        B'11111101'    ; Prescaler TMR2 = 1:4
      banksel      T2CON
      movwf        T2CON
      movlw        B'11111111'    ; Number in register PR2
      banksel      PR2
      movwf        PR2

      banksel      CCP1CON
      movlw        B'00001100'    ; Bits to configure CCP1 module
      movwf        CCP1CON
loop
      goto         loop           ; Remain here
      end                         ; End of program
```

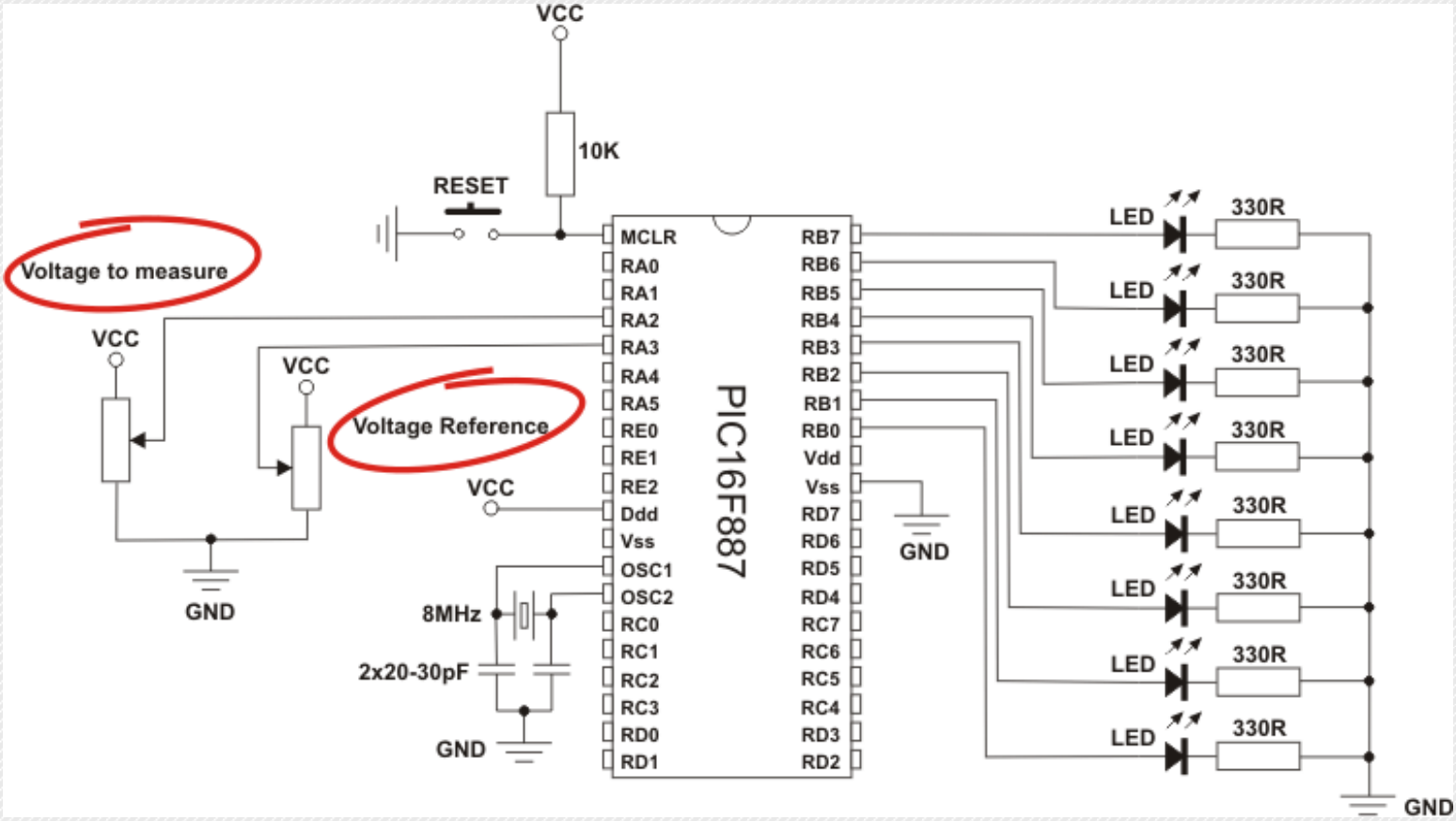
EXAMPLE 11

Using A/D converter

PIC16F887 A/D converter is used in this example. Everything is quite simple. A variable analog signal is applied on the AN2 pin while the result of conversion is shown on port B as a binary number. In order to simplify the program as much as

possible, only 8 lower bits of the result of conversion are shown. GND is used as a negative voltage reference Vref-, while positive voltage reference is applied on the AN3 pin. It enables voltage measurement scale to "stretch and shrink".

To make this clear, the A/D converter always generates a 10-bit binary result, which means that it detects a total of 1024 voltage levels (210=1024). The difference between two voltage levels is not always the same. The less the difference between Vref+ and Vref, the less the difference will be between two of 1024 levels. Accordingly, the A/D converter is able to detect slight changes in voltage.



Example 11:

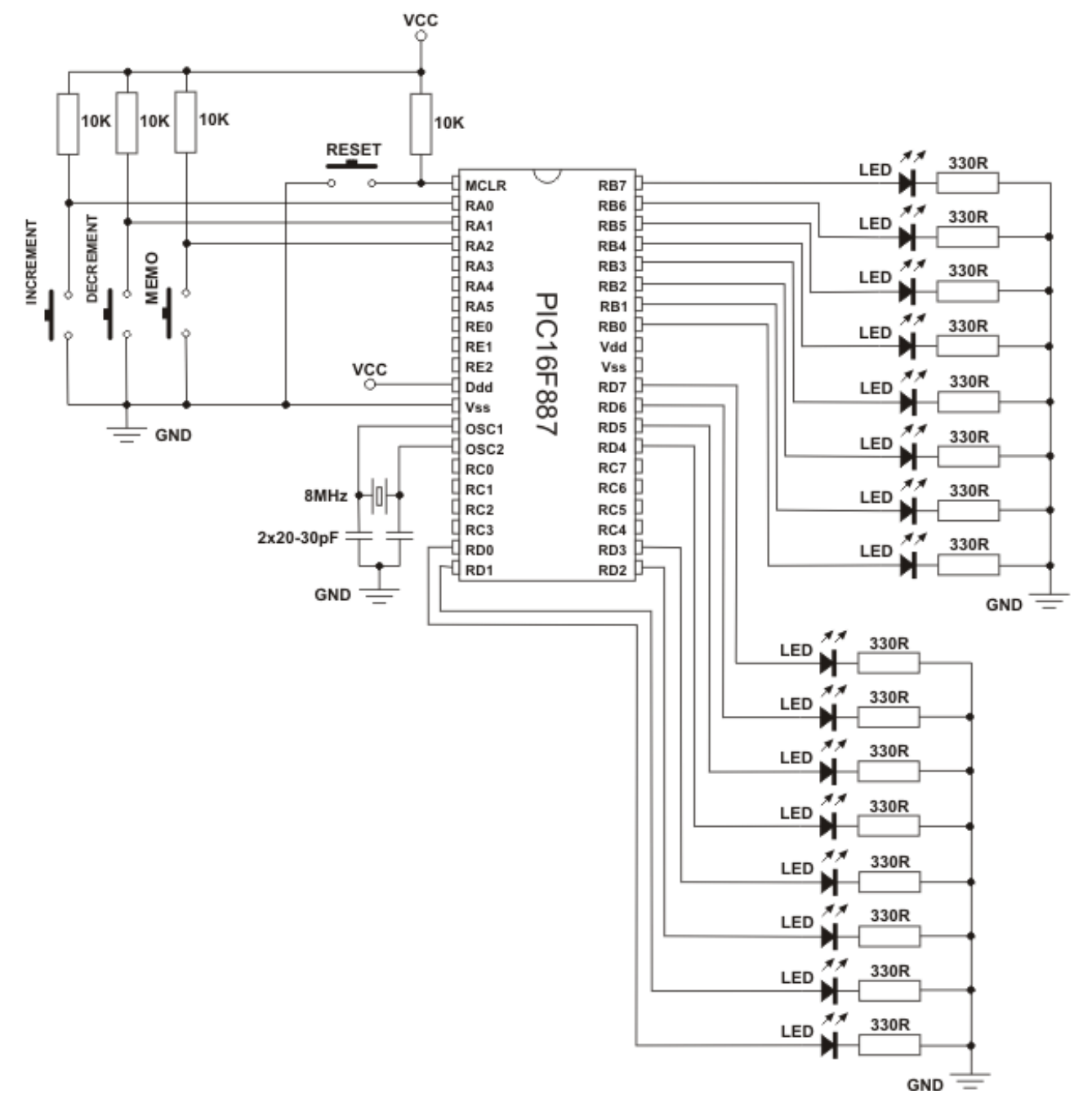
```
;********************** Header ********************************************
;*********************** PROGRAM START ************************************

       org         0x0000          ; Address of the first program instruction

       banksel     TRISB           ; Selects bank containing register TRISB
       clrf        TRISB           ; All port B pins are configured as outputs
       movlw       B'00001100'
       movwf       TRISA           ; Pins RA2 and RA3 are configured as inputs

       banksel     ANSEL           ; Selects bank containing register ANSEL
       movlw       B'00001100'     ; Inputs AN2 and AN3 are analog while
       movwf       ANSEL           ; all other pins are digital
       clrf        ANSELH

       banksel     ADCON1          ; Selects bank including register ADCON1
       bsf         ADCON1,ADFM     ; Right justification of result
       bcf         ADCON1,VCFG1    ; Voltage Vss is used as Vref
       bsf         ADCON1,VCFG0    ; RA3 pin voltage is used as Vref+

       banksel     ADCON0          ; Selects bank containing register ADCON0
       movlw       B'00001001'     ; AD converter uses clock Fosc/2, AD channel
       movwf       ADCON0          ; on RA2 pin is used for conversion and
                                   ; AD converter is enabled
loop
       banksel     ADCON0
       btfsc       ADCON0,1        ; Tests bit GO/DONE
       goto        loop            ; Conversion in progress, remain in
```

```
                                ; loop
        banksel     ADRESL
        movf        ADRESL,w    ; Lower byte of conversion result is
                                ; copied to W
        banksel     PORTB
        movwf       PORTB       ; Byte is copied to PORTB
        bsf         ADCON0,1    ; Starts new conversion
        goto        loop        ; Jump to label "loop"
        end                     ; End of program
```

EXAMPLE 12

Using EEPROM memory

This example demonstrates write to and read from built-in EEPROM memory. The program works as follows. The main loop constantly reads EEPROM memory location at address 5 (decimal). This number is displayed on port D. The same loop tests the state of three push-buttons connected to port A. The push-buttons "INCREMENT" and "DECREMENT" have the same purpose like in example 7 - increment and decrement the variable "cnt" which is thereafter displayed on port B. The push-button "MEMO" enables that variable to be written to EEPROM memory. In order to check it, it is enough to press this push-button and switch off the device. On the next switch on, the program displays the value of the variable on port D (at the moment of writing, this value was displayed on port B).



Example 12:

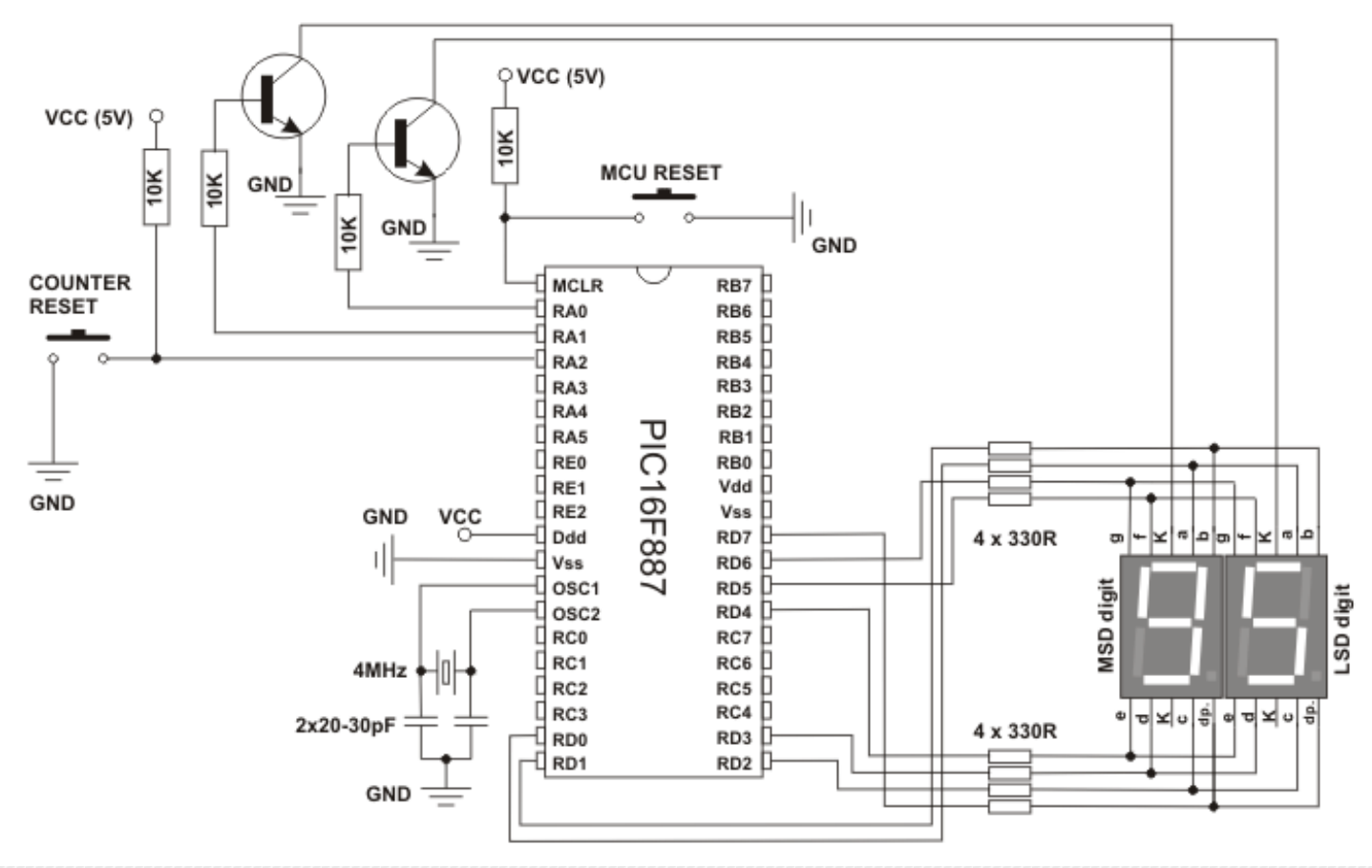
```
;********************** Header ********************************************
;************ Defining variables in program ********************************
       cblock      0x20            ; Block of variables starts at address 20h

       HIcnt
       LOcnt
       LOOPcnt
       cnt
       endc                        ; End of block

;*******************************************************************************
       ORG         0x000           ; Reset vector
       nop
       goto        main            ; Go to start of the program (label "main")
;*******************************************************************************
       include     "pause.inc"
       include     "button.inc"
;*******************************************************************************
main
       banksel     ANSEL           ; Selects bank containing ANSEL
       clrf        ANSEL
       clrf        ANSELH          ; All pins are digital

       banksel     TRISB
       bsf         TRISA, 0        ; Input pin
       bsf         TRISA, 1        ; Input pin
       bsf         TRISA, 2        ; Input pin
       clrf        TRISB           ; All port B pins are outputs
       clrf        TRISD           ; All port D pins are outputs
       banksel     PORTB
       clrf        PORTB           ; PORTB=0
       clrf        PORTD           ; PORTD=0
       clrf        cnt             ; cnt=0
Loop
       banksel     PORTA
       button      PORTA,0,0,Increment
       button      PORTA,1,0,Decrement
       button      PORTA,2,0,Save

       banksel     EEADR
       movlw       .5              ; Reads EEPROM memory location
       movwf       EEADR           ; at address 5
       banksel     EECON1
       bcf         EECON1,EEPGD
       bsf         EECON1,RD       ; Reads data from EEPROM memory
       banksel     EEDATA
       movfw       EEDATA          ; Moves data to W
       banksel     PORTD
       movwf       PORTD           ; Data is moved from W to PORTD
       goto        Loop
Increment                          ; Increments number on port B
       incf        cnt, f
       movf        cnt, w
       movwf       PORTB
       goto        Loop
Decrement                          ; Decrements number on port B
       decf        cnt, f
       movf        cnt, w
       movwf       PORTB
       goto        Loop
Save                               ; Copies data from port B to EEPROM
       banksel     EEADR           ; memory location at address 5
       movlw       .5
       movwf       EEADR           ; Writes address
       banksel     PORTB
       movfw       PORTB           ; Copies port B to register W
       banksel     EEDAT
```

```
        movwf       EEDAT           ; Writes data to temporary register
        banksel     EECON1
        bcf         EECON1,EEPGD
        bsf         EECON1,WREN     ; Write enabled

        bcf         INTCON,GIE      ; All interrupts disabled
        btfsc       INTCON,GIE
        goto        $-2

        movlw       55h
        movwf       EECON2
        movlw       H'AA'
        movwf       EECON2
        bsf         EECON1,WR

        btfsc       EECON1,WR       ; Wait for write to complete
        goto        $-1

        bsf         INTCON,GIE      ; Interrupt enabled
        bcf         EECON1,WREN
        goto        Loop            ; Tests push-buttons again
        end                         ; End of program
```

EXAMPLE 13

Two-digit LED counter, multiplexing

In this example, the microcontroller operates as a two-digit counter. Concretely, the variable Dval is decremented (slow enough to be visible) and its value is displayed on twodigit LED display (99-0). The challenge is to enable binary number to be converted in decimal one and split it in two digits (tens and ones). Besides, since the LED display segments are connected in parallel, it is necessary to ensure that they change fast in order to make impression of simultaneous light emission (time-division multiplexing). Remember that in electronics, multiplexing allows several analog signals to be processed by one analog-todigital converter (ADC). In this very case, time-division multiplexing is performed by the timer TMR0, while binary to decimal number conversion is performed in macro "digbyte". Counter may be reset to its starting value (99) at any moment by pressing the pushbutton "COUNTER RESET".



Example 13:

```
;************************ Header ********************************************
;****************************************************************************
;                    DEFINING VARIABLES IN PROGRAM
       w_temp      EQU 0x7D        ; Variable for saving W register

       status_temp EQU 0x7E        ; Variable for saving STATUS register

       pclath_temp EQU 0x7F        ; Variable for saving PCLATH register

       CBLOCK      0x20            ; Block of variables starts at address 20h

       Digtemp
       Dig0                        ; Variables for displaying digits - LSB
       Dig1
       Dig2
       Dig3                        ; Variables for displaying digits - MSB
       Dval                        ; Counter value
       One                         ; Auxiliary variable which determines which
                                   ; display is to be switched on

       ENDC                        ; End of block of variables

       poc_vr      EQU .99         ; Initial  counter value is 99

       include     "Digbyte.inc"

;****************************************************************************
       ORG         0x0000          ; First instruction address
       goto        main            ; Jump to label "main"
;****************************************************************************
       ORG         0x0004          ; Interrupt vector address

       movwf       w_temp          ; Move w register to w_temp register


       movf        STATUS,w        ; Move STATUS register to status_temp
       movwf       status_temp     ; register

       movf        PCLATH,w        ; Move PCLATH register to pclath_temp
       movwf       pclath_temp     ; register

       ; Start of interrupt routine...

       BANKSEL     TMR0
       movlw       .100
       movwf       TMR0
       bcf         INTCON, T0IF

       bcf         PORTA, 0
       bcf         PORTA, 1
       btfsc       One, 0
       goto        Lsdon
       goto        Msdon

Lsdon
       incf        One, f
       movlw       HIGH (Bcdto7seg)
       movwf       PCLATH
       digbyte     Dval
       movf        Dig1, w
       call        Bcdto7seg       ; Place L1 mask on the PORTD
       movwf       PORTD
       bsf         PORTA, 1
       goto        ISR_end

Msdon
       incf        One, f
       movlw       HIGH (Bcdto7seg)
       movwf       PCLATH
```

```
        digbyte     Dval
        movf        Dig0, w
        call        Bcdto7seg       ; Place LO mask on the PORTD
        movwf       PORTD
        bsf         PORTA, 0
        goto        ISR_end

        ; End of interrupt routine...

ISR_end
        movf        pclath_temp,w   ; PCLATH register is given its original
        movwf       PCLATH          ; state

        movf        status_temp,w   ; STATUS register is given its original
        movwf       STATUS          ; state

        swapf       w_temp,f        ; W register is given its original
                                    ; state
        swapf       w_temp,w
        retfie                      ; Return from interrupt routine
main
        banksel     ANSEL           ; Selects bank containing ANSEL
        clrf        ANSEL           ; All pins are digital
        clrf        ANSELH

        BANKSEL     TRISA
        movlw       b'11111100'     ; RA0 and RA1 are configured as outputs and
                                    ; used for 7-segment display multiplexing
                                    ; RA2 is input push-button for initializa
                                    ; tion

        movwf       TRISA
        clrf        TRISD

        BANKSEL     OPTION_REG
        movlw       b'10000110'     ; TMR0 is incremented each 32us (Fclk=8MHz)
        movwf       OPTION_REG

        BANKSEL     PORTA
        movlw       poc_vr
        movwf       Dval            ; Dval contains counter value
        movlw       b'00000001'     ; Initializes variable specifying display
        movwf       One             ; to switch on
        movwf       PORTA
        movlw       .100
        movwf       TMR0            ; TMR0 interrupt appr.every 10ms
        bsf         INTCON, GIE     ; Global interrupt enabled
        bsf         INTCON, T0IE    ; Timer TMR0 interrupt enabled
        bcf         INTCON, T0IF
Loop
        btfss       One, 3          ; Falling edge encountered?
        goto Dec                    ; Yes! Go to Dec
        btfss       PORTA, 2        ; Counter reset button pressed?
        goto        Reset           ; Yes! Go to Reset
        goto        Loop
        ; Decrement Dval counter by 1
Dec
        btfss       One, 3
        goto        Dec
        movf        Dval, f
        btfsc       STATUS, Z       ; Is Dval equal to 0?
        goto        Loop            ; If it is, go to loop and wait for T2
        decf        Dval, f         ; If Dval not equal to 0, decrement it by 1
        goto        Loop
Reset
        btfss       PORTA, 2        ; Wait for rising edge
        goto        Reset
        movlw       poc_vr
        movwf       Dval            ; Write initial value to counter
        goto        Loop
;*******************************************************************************
        ORG         0x0300          ; Lookup table is at the top of third page, but
                                    ; can be placed at some other place, it is impor
```

```
                                        ; tant to have it all on one page
Bcdto7seg
       addwf       PCL, f
       DT          0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f
;*****************************************************************************
       END         ; End of program
```

Macro "digbyte":

```
digbyte MACRO arg0
       LOCAL       Exit0
       LOCAL       Exit1
       LOCAL       Exit2

       clrf        Dig0
       clrf        Dig1
       clrf        Dig2
       clrf        Dig3

       movf        arg0, w
       movwf       Digtemp
       movlw       .100
Exit2
       incf        Dig2, f
       subwf       Digtemp, f
       btfsc       STATUS, C
       goto        Exit2
       decf        Dig2, f
       addwf       Digtemp, f
Exit1
       movlw       .10
       incf        Dig1, f
       subwf       Digtemp, f
       btfsc       STATUS, C
       goto        Exit1
       decf        Dig1, f
       addwf       Digtemp, f
Exit0
       movf        Digtemp, w
       movwf       Dig0
       ENDM
```

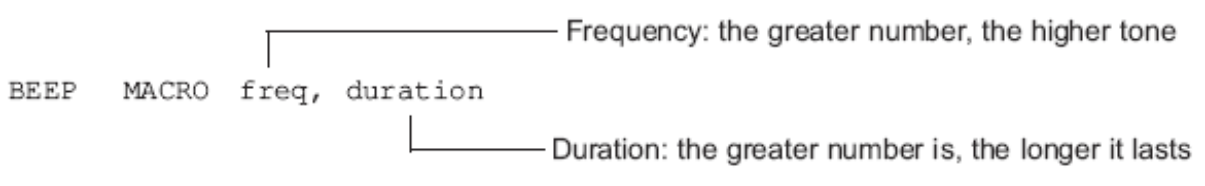
Macro digbyte is used to convert the number from digital to decimal format. Besides, digits of such decimal number are stored into special registers in order to enable them to be displayed on LED displays.

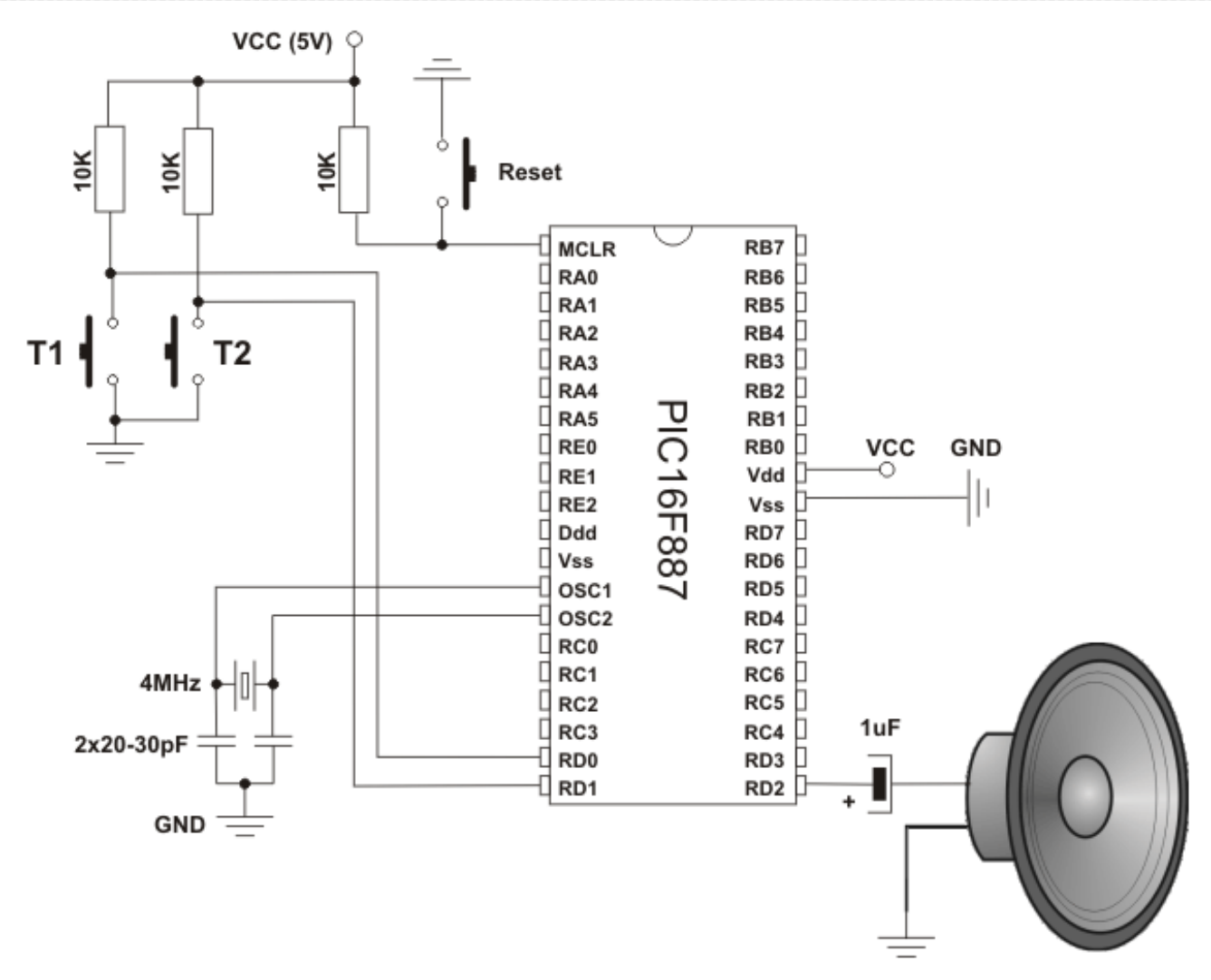
EXAMPLE 14

Sound generating, using macros

The generation of sound is a task commonly assigned to the microcontroller. Basically, it all comes to generating a pulse sequence on one output pin. While doing so, the proportion of logic zero (0) to logic one (1) duration determines the tone pitch and by changing different tones, different melodies arise.

In this example, any press on push-buttons T1 and T2 generates a sound. The appropriate instructions are stored in macro "beep" containing two arguments.





Example 14:

```
;********************** Header ***************************************
;******************* Defining variables in program ********************
       cblock      0x20
       HIcnt                       ; Auxiliary variables for macro pausems
       LOcnt
       LOOPcnt
       PRESCwait
       Beep_TEMP1                  ; Belongs to macro "BEEP"
       Beep_TEMP2
       Beep_TEMP3
       endc

#define  BEEPport PORTD, 2         ; Speaker pin
#define  BEEPtris TRISD, 2

       expand

;*************************************************************************
       ORG         0x0000          ; RESET vector address
       goto        main            ; Jump to program start (label - main)
;*************************************************************************
; remaining code goes here

       include     "pause.inc"
       include     "button.inc"
       include     "beep.inc"
main
       banksel     ANSEL           ; Selects bank containing ANSEL
       clrf        ANSEL           ; All outputs are digital
```

```
        clrf        ANSELH

        banksel     TRISD
        movlw       b'11111011'     ; PORTA D initialization
        movwf       TRISD
        banksel     PORTD
        BEEPinit                    ; Macro "Beep"
Loop
        button      PORTD,0,0,Play1 ; Push-button 1
        button      PORTD,1,0,Play2 ; Push-button 2
        goto        Loop

Play1                               ; First tone
        BEEP        0xFF, 0x02
        BEEP        0x90, 0x05
        BEEP        0xC0, 0x03
        BEEP        0xFF, 0x03
        goto        Loop

Play2                               ; Second tone
        BEEP        0xBB, 0x02
        BEEP        0x87, 0x05
        BEEP        0xA2, 0x03
        BEEP        0x98, 0x03
        goto        Loop
;*****************************************************************
        END                         ; End of program
```

Macro "beep":

```
BEEPinit    MACRO
        bcf         STATUS, RP0
        bcf         STATUS, RP1
        bcf         BEEPport
        bsf         STATUS, RP0
        bcf         STATUS, RP1
        bcf         BEEPtris
        movlw       b'00000111'     ; TMR0 prescaler rate 1:256
        movwf       OPTION_REG      ; OPTION <- W
        bcf         STATUS, RP0
        bcf         STATUS, RP1
        ENDM
BEEP    MACRO       freq, duration
        bcf         STATUS, RP0
        bcf         STATUS, RP1
        movlw       freq
        movwf       Beep_TEMP1
        movlw       duration
        movwf       Beep_TEMP2
        call        BEEPsub
        ENDM
;*****************************************************************
; Subroutines

BEEPsub
        clrf        TMR0            ; Counter initialization
        bcf         INTCON, T0IF
        bcf         BEEPport
BEEPa
        bcf         INTCON, T0IF    ; Clears TMR0 Overflow Flag
BEEPb
        bsf         BEEPport
        call        B_Wait          ; Logic one "1" duration
        bcf         BEEPport
        call        B_Wait          ; Logic zero "0" duration
        btfss       INTCON, T0IF    ; Check TMR0 Overflow Flag,
        goto        BEEPb           ; skip next if set
        decfsz      Beep_TEMP2, f   ; Is Beep_TEMP2 = 0 ?
        goto BEEPa                  ; Go to BEEPa again
        return
```

```
B_Wait
       movf Beep_TEMP1, w
       movwf Beep_TEMP3
B_Waita
       decfsz Beep_TEMP3, f
       goto B_Waita
       return
```

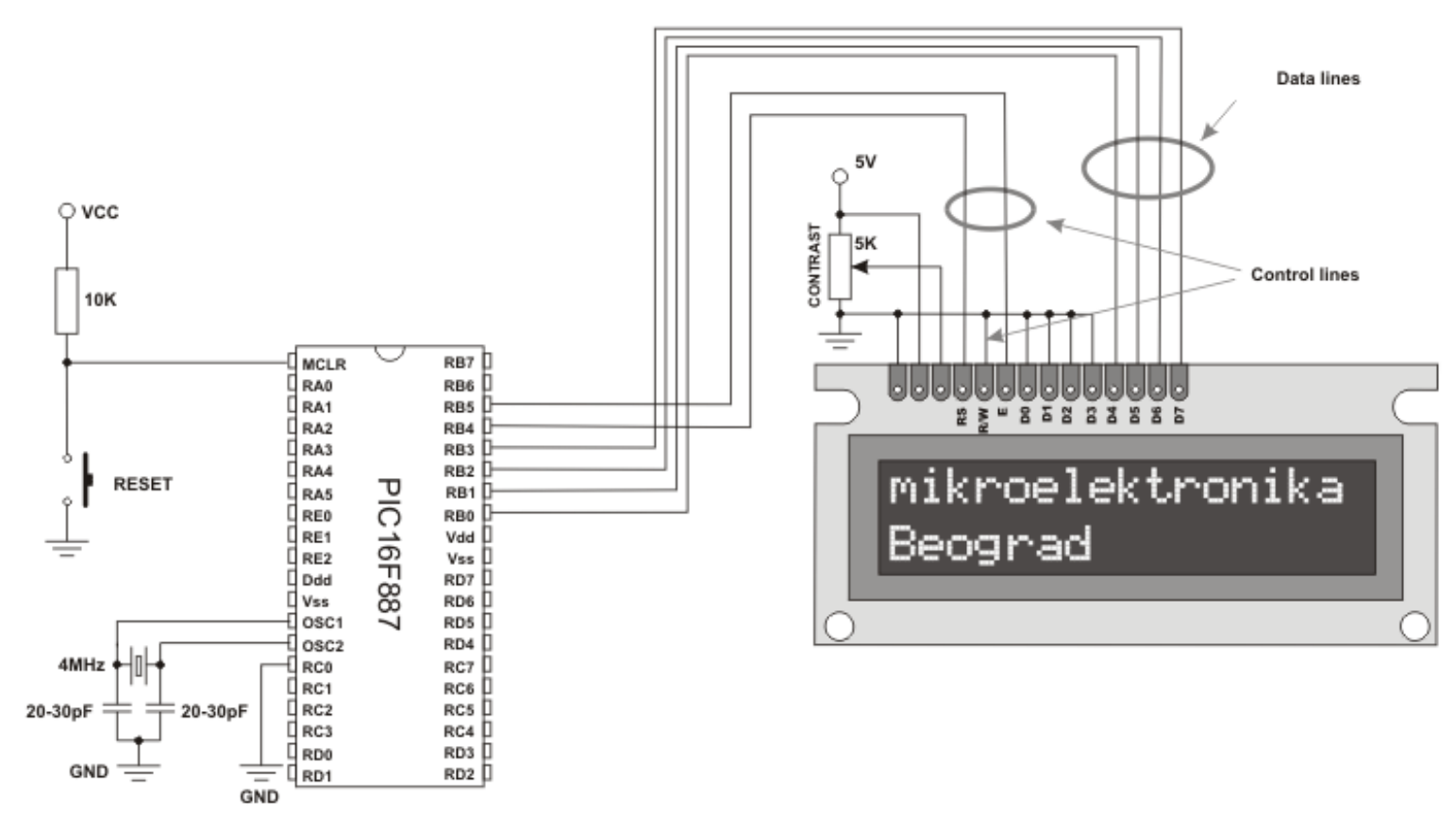
EXAMPLE 15

Using LCD display

This example illustrates the use of the alphanumeric LCD display. The program itself is very simple because macros are used (usually the effort of creating Macros pays off in the end).

Two messages written on two lines change on display. The second message is intended to display the current temperature. Since no sensor is installed, the measurement is not really carried out, the variable "temp" appears on the display instead of the measured temperature.

In reality, the current temperature or some other measured value would be displayed.



Example 15:

```
;********************** Header ************************************
                ; DEFINING VARIABLES IN PROGRAM

       CBLOCK      0x20            ; Block of variables starts at address 20h

       HIcnt                       ; Belongs to macro "pausems"
       LOcnt
       LOOPcnt

       LCDbuf                      ; Belongs to functions "LCDxxx"
       LCDtemp
```

```
        LCDportBuf                  ; LCD Port Buffer

        Digtemp                     ; Belongs to macro "digbyte"
        Dig0
        Dig1
        Dig2
        Dig3

        temp
        ENDC                        ; End of block

LCDport   EQU PORTB   ; LCD is on PORTB (4 data lines on RB0-RB3)
RS        EQU 4       ; RS line connected to RB4
EN        EQU 5       ; EN line connected to RB5

;*****************************************************************
        ORG         0x0000          ; Reset vector address
        nop
        goto        main            ; Go to beginning of the program (label "main")
;*****************************************************************
        include    "LCD.inc"
        include    "digbyte.inc"
        include    "pause.inc"
;*****************************************************************
main
        banksel     ANSEL           ; Selects bank containing ANSEL
        clrf        ANSEL           ; All pins are digital
        clrf        ANSELH

        bcf         STATUS,RP0      ; Bank0 active only
        bcf         STATUS,RP1
        movlw       .23
        movwf       temp            ; Move arbitrary value to variable
                                    ; is to be displayed on LCD
        lcdinit                     ; LCD initialization
Loop
        lcdcmd      0x01            ; Instruction to clear LCD
        lcdtext     1, "mikroelektronika" ; Write text from the begin
                                          ; ning of the first line
        lcdtext     2, "Beograd"  ; Write text from the beginning of
                                  ; the second line
        pausems     .2000           ; 2 sec. delay
        lcdcmd      0x01            ; Instruction to clear LCD
        lcdtext 1, "Temperatura1" ; Write text from the begin
                                  ; ning of the first line
        lcdtext 2, "temp=" ; Write text from the beginning of
                           ; the second line
        lcdbyte     temp            ; Write variable (dec.)
        lcdtext 0, " C"           ; Write text after cursor
        pausems     .2000           ; 2 sec. delay
        goto        Loop
;*****************************************************************
        end                         ; End of program
```

LCD.inc

```
;*****************************************************************
; Initialization must be done by using macro lcdinit before access
; ing LCD
;*****************************************************************
lcdinit    MACRO
        bcf         STATUS, RP0     ; Bank0
        bcf         STATUS, RP1
        clrf        LCDportBuf
        movf        LCDportBuf, w
        movwf       LCDport
        bsf         STATUS, RP0     ; Bank1
        bcf         STATUS, RP1
        clrf        TRISB           ; LCDport with output LCD
        bcf         STATUS, RP0     ; Bank0
```

```
        bcf         STATUS, RP1

; Function set (4-bit mode change)
        movlw       b'00100000'
        movwf       LCDbuf
        swapf       LCDbuf, w
        movwf       LCDportBuf
        bcf         LCDportBuf, RS
        movf        LCDportBuf, w
        movwf       LCDport
        bsf         LCDportBuf, EN
        movf        LCDportBuf, w
        movwf       LCDport
        bcf         LCDportBuf, EN
        movf        LCDportBuf, w
        movwf       LCDport
        call        Delay1ms        ; 1 ms delay

; Function set (display mode set)
        lcdcmd      b'00101100'
        call        Delay1ms        ; 1 ms delay

; Display ON/OFF Control
        lcdcmd      b'00001100'
        call        Delay1ms        ; 1 ms delay

; Entry Mode Set
        lcdcmd      b'00000110'
        call        Delay1ms        ; 1 ms delay

; Display Clear
        lcdcmd      b'00000001'
        pausems     .40             ; 40 ms delay

; Function set (4-bit mode change)
        movlw       b'00100000'
        movwf       LCDbuf
        swapf       LCDbuf, w
        movwf       LCDportBuf
        bcf         LCDportBuf, RS
        movf        LCDportBuf, w
        movwf       LCDport
        bsf         LCDportBuf, EN
        movf        LCDportBuf, w
        movwf       LCDport
        bcf         LCDportBuf, EN
        movf        LCDportBuf, w
        movwf       LCDport
        call        Delay1ms        ; 1 ms delay

; Function set (display mode set)
        lcdcmd      b'00101100'
        call        Delay1ms        ; 1 ms delay

; Display ON/OFF Control
        lcdcmd      b'00001100'
        call        Delay1ms        ; 1 ms delay

; Entry Mode Set
        lcdcmd      b'00000110'
        call        Delay1ms        ; 1 ms delay

; Display Clear
        lcdcmd      b'00000001'
        pausems     .40             ; 40 ms delay

        ENDM

;*****************************************************************
; lcdcmd sends command to LCD (see the table on the previous page)
; lcdclr is the same as lcdcmd 0x01
;*****************************************************************
lcdcmd MACRO LCDcommand         ; Send command to LCD
```

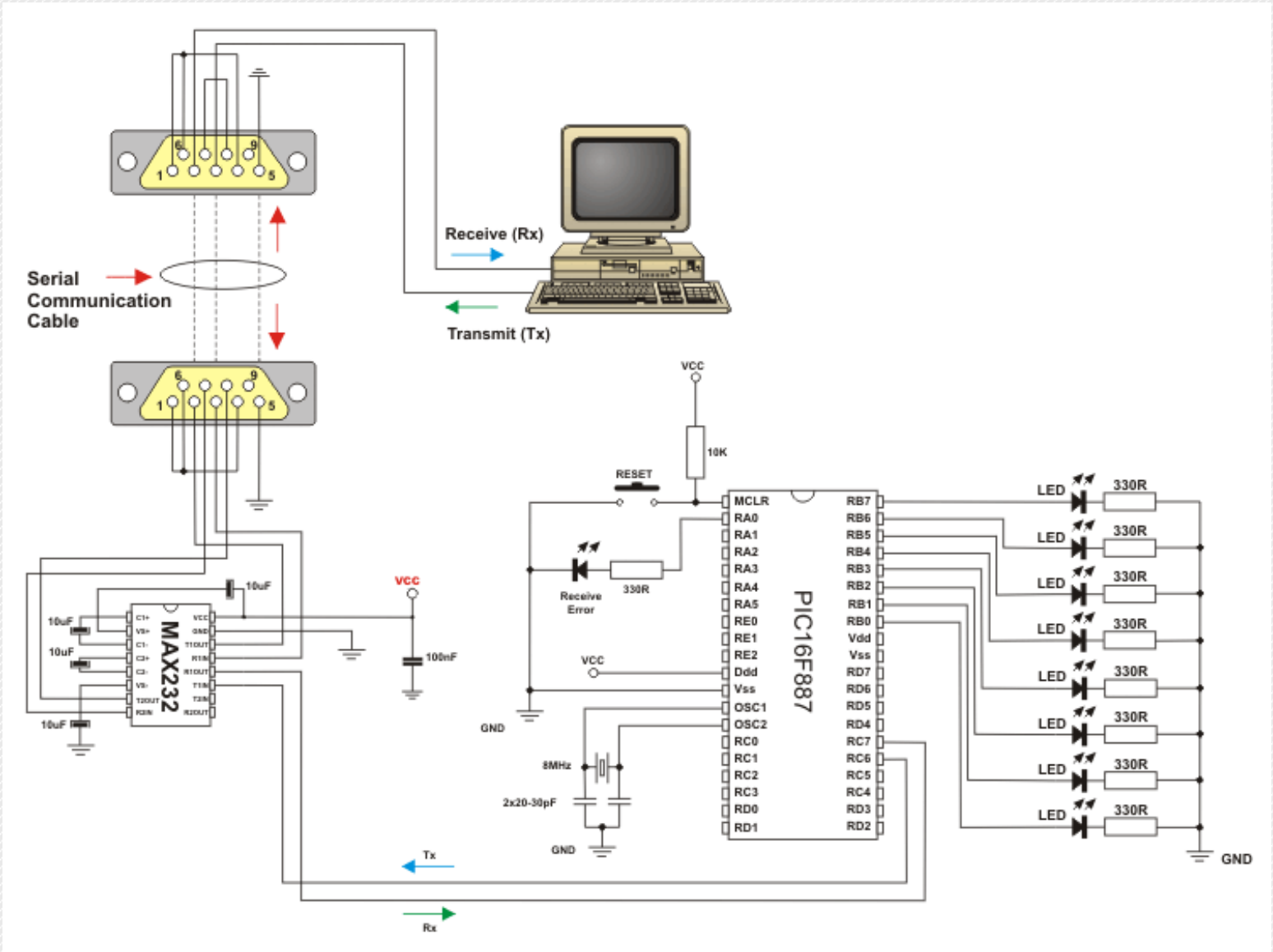
```
        movlw       LCDcommand
        call        LCDcomd
        ENDM

LCDcomd
        movwf       LCDbuf
        bcf         LCDportBuf, RS
        movf        LCDportBuf, w
        movwf       LCDport
        goto        LCDwr
LCDdata
        movwf       LCDbuf
        bsf         LCDportBuf, RS
        movf        LCDportBuf, w
        movwf       LCDport
        goto        LCDwr
LCDwr
        swapf       LCDbuf, w
        call        SendW
        movf        LCDbuf, w
        call        SendW
        return
SendW
        andlw       0x0F
        movwf       LCDtemp

        movlw       0xF0
        andwf       LCDportBuf, f
        movf        LCDtemp, w
        iorwf       LCDportBuf, f
        movf        LCDportBuf, w
        movwf       LCDport
        call        Delay1ms
        bsf         LCDportBuf, EN
        movf        LCDportBuf, w
        movwf       LCDport
        bcf         LCDportBuf, EN
        movf        LCDportBuf, w
        movwf       LCDport
        call        Delay1ms
        return

;*****************************************************************
; lcdtext writes text containing 16 characters which represents a
; macro argument. The first argument select selects the line in which
; text writing is to start. If select is 0, text writing starts from
; cursor current position.
;*****************************************************************
lcdtext    MACRO select, text      ; This macro writes text from cursor
                                   ; current position. Text is specified
                                   ; in argument consisting of 16 charac
                                   ; ters
        local       Message
        local       Start
        local       Exit
        local       i=0
        goto        Start
        Message     DT text        ; Create lookup table from arguments
        DT          0
Start
        IF (select == 1)
        lcdcmd b'10000000'
        ELSE
        IF (select == 2)
        lcdcmd b'11000000'
        ENDIF
        ENDIF

        WHILE (i<16)               ; Repeat conditional program compiling 16 times
        call Message+i             ; Read lookup table and place value in W
        addlw 0
        bz Exit                    ; until 0 is read
        call LCDdata               ; Call routine displaying W on LCD
```

```
        i=i+1
        ENDW
Exit
        ENDM

;*****************************************************************
; This macro writes value in size of 1 byte on LCD
; excluding leading zeros
;*****************************************************************
lcdbyte    MACRO arg0
        digbyte     arg0            ; A hundred is in Dig2,
                                    ; A ten is in Dig1 and one in Dig0
        movf        Dig2, w
        addlw       0x30
        call        LCDdata
        movf        Dig1, w         ; If digit is 0 move cursor
        addlw       0x30
        call        LCDdata
        movf        Dig0, w         ; If digit is 0 move cursor
        addlw       0x30
        call        LCDdata
        ENDM
;*****************************************************************
; 1ms Delay
Delay1ms:
        movlw       .200
        movwf       LOOPcnt
Delay10us:
        nop                         ;1us
        nop                         ;1us
        nop                         ;1us
        nop                         ;1us
        nop                         ;1us
        nop                         ;1us
        nop                         ;1us
        decfsz      LOOPcnt, f      ;1us
        goto        Delay10us       ;2us

        return
```

EXAMPLE 16

RS232 serial communication

This example illustrates the use of the microcontroller's EUSART module. Connection to the PC is enabled through RS232 standard. The program works in the following way: Every byte received via the serial communication is displayed using LED diodes connected to port B and is automatically returned to the transmitter thereafter. If an error occurs on receive, it will be signalled by switching the LED diode on. The easiest way to test device operation in practice is by using a standard Windows program called *Hyper Terminal*.



Example 16:

```
;********************** Header ***************************************
;                  DEFINING VARIABLES IN PROGRAM

       w_temp      EQU 0x7D        ; Variable for saving W register
       status_temp EQU 0x7E        ; Variable for saving STATUS register
       pclath_temp EQU 0x7F        ; Variable for saving PCLATH w register

       cblock      0x20            ; Block of variables starts at address 20 h
       Port_A                      ; Variable at address 20 h
       Port_B                      ; Variable at address 21 h
       RS232temp                   ; Variable at address 22 h
       RXchr                       ; Variable at address 23 h
       endc                        ; End of block of variables
;*************************************************************************
       ORG         0x0000          ; Reset vector
       nop
       goto        main            ; Go to beginning of program (label "main")
;*************************************************************************
       ORG         0x0004          ; Interrupt vector address

       movwf       w_temp          ; Save value of W register
       movf        STATUS,w        ; Save value of STATUS register
       movwf       status_temp
       movf        PCLATH,w        ; Save value of PCLATH register
       movwf       pclath_temp
;*************************************************************************
```

```
; This part of the program is executed in interrupt routine
       banksel     PIE1
       btfss       PIE1, RCIE
       goto        ISR_Not_RX232int
       banksel     PIE1
       btfsc       PIR1, RCIF
       call        RX232_int_proc

ISR_Not_RX232int
       movf        pclath_temp,w
       movwf       PCLATH          ; PCLATH is given its original value

       movf        status_temp,w
       movwf       STATUS          ; STATUS is given its original value
       swapf       w_temp,f
       swapf       w_temp,w        ; W is given its original value

       retfie                      ; Return from interrupt routine
;***********************************************************************
RX232_int_proc                     ; Check if error has occurred
       banksel     RCSTA
       movf        RCSTA, w
       movwf       RS232temp
       btfsc       RS232temp, FERR
       goto        RX232_int_proc_FERR
       btfsc       RS232temp, OERR
       goto        RX232_int_proc_OERR
       goto        RX232_int_proc_Cont

RX232_int_proc_FERR
       bcf         RCSTA, CREN     ; To clear FERR bit, receiver is first
                                   ; switched off and on afterwards
       nop                         ; Delay ...
       nop
       bsf         RCSTA, CREN
       movf        RCREG, w        ; Reads receive register and clears FERR bit
       bsf         Port_A, 0       ; Switches LED on ( UART error indicator)
       movf        Port_A, w
       movwf       PORTA
       goto        RS232_exit

RX232_int_proc_OERR
       bcf         RCSTA, CREN     ; Clears OERR bit
       nop                         ; Delay ...
       nop
       bsf         RCSTA, CREN
       movf        RCREG, w        ; Reads receive register and clears FERR bit
       bsf         Port_A, 1       ; Switches LED on ( UART error indicator)
       movf        Port_A, w
       movwf       PORTA
       goto        RS232_exit

RX232_int_proc_Cont
       movf        RCREG, W        ; Reads received data
       movwf       RXchr
       movwf       PORTB
       movwf       TXREG           ; Sends data back to PC

RS232_exit
       return                      ; Return from interrupt routine
;***********************************************************************
; Main program

main
       banksel     ANSEL           ; Selects bank containing ANSEL
       clrf        ANSEL           ; All inputs are digital
       clrf        ANSELH

       ;--------------------------------------
       ; Port configuration
       ;--------------------------------------
       banksel     TRISA
       movlw       b'11111100'
```

```
        movwf       TRISA
        movlw       b'00000000'
        movwf       TRISB
        ;--------------------------------------
        ; Setting initial values
        ;--------------------------------------
        banksel     PORTA
        movlw       b'11111100'
        movwf       PORTA
        movwf       Port_A
        movlw       b'00000000'
        movwf       PORTB
        movwf       Port_B
        ;--------------------------------------
        ; USART - setting for 38400 bps
        ;--------------------------------------
        banksel     TRISC
        bcf         TRISC, 6        ; RC6/TX/CK = output
        bsf         TRISC, 7        ; RC7/RX/DT = input

        banksel     BAUDCTL
        bsf         BAUDCTL, BRG16
        banksel     SPBRG
        movlw       .51             ; baud rate = 38400
                                    ; ( Fosc/(4*(SPBRG+1)) ) Error +0.16%
        movwf       SPBRG
        clrf        SPBRGH

        banksel     TXSTA
        bcf         TXSTA, TX9      ; Data is 8-bit wide
        bsf         TXSTA, TXEN     ; Data transmission enabled
        bcf         TXSTA, SYNC     ; Asynchronous mode
        bsf         TXSTA, BRGH     ; High-speed Baud rate

        banksel     RCSTA
        bsf         RCSTA, SPEN     ; RX/DT and TX/CK outputs configuration
        bcf         RCSTA, RX9      ; Select mode for 8-bit data receive
        bsf         RCSTA, CREN     ; Receive data enabled
        bcf         RCSTA, ADDEN    ; No address detection, ninth bit may be
                                    ; used as parity bit
        movf        RCSTA, W
        movf        RCREG, W
        ;--------------------------------------
        ; Interrupts enabled
        ;--------------------------------------
        banksel     PIE1
        bsf         PIE1, RCIE      ; USART Rx interrupt enabled

        bsf         INTCON, PEIE    ; All peripheral interrupts enabled
        bsf         INTCON, GIE     ; Global interrupt enabled

        ;--------------------------------------
        ; Remain here
        ;--------------------------------------
        goto $

        end                         ; End of program
```

Previous Chapter | Table of Contents | Next Chapter

Appendix C: Development Systems

How to start working?

A microcontroller is a good-natured "genie in the bottle" and no extra knowledge is required to use it.

In order to create a device controlled by the microcontroller, it is necessary to provide the simplest PC, program for compiling and simple device to transfer that code from PC to chip itself.

Even though this process is quite logical, there are often some queries, not because it is complicated, but for numerous variations. Let's take a look...

WRITING PROGRAM IN ASSEMBLY LANGUAGE

In order to write a program for the microcontroller, a specialized program in the Windows environment may be used. Any program for text processing can be used for this purpose. The point is to write all instructions in such an order they should be executed by the microcontroller, observe the rules of assembly language and write instructions exactly as they are defined. In other words, you just have to follow the program idea! That's all! When using custom software, there are numerous tools which are also installed to aid in the development process. One such tool is the Simulator. This enables the user to test the code prior to burning it to the MCU.

```
Loop      button PORTA,0,0,Increment
          button PORTA,1,0,Decrement
```

```
            goto Loop

Increment incf cnt,f
            movf cnt,w
            movwf PORTB
            goto Loop

Decrement decf cnt,f
            movf cnt,w
            movwf PORTB
```

To enable the compiler to perform its task successfully, it is necessary that a document containing this program has the extension, .asm in its name, for example: Program.asm

When a specialized program (MPLAB) is used, this extension will be automatically added. If any other program for text processing (Notepad) is used then the document should be saved and renamed. For example: Program.txt -> Program.asm.

Note for lazy ones: skip this procedure, open a new .asm document in MPLAB and simply copy/paste the text of the program written in assembly language.

COMPILING PROGRAM

The microcontroller does not understand assembly language as such. This is why it is necessary to compile the program into machine language. It is more than simple when using a specialized program (MPLAB) because a compiler is part of the software! Just one click on the appropriate icon solves the problem and a new document with .hex extension pops out. It is actually the same program, but compiled into computer language which the microcontroller perfectly understands. Such documentation is commonly named "hex code" and seemingly represents a meaningless sequence of numbers in hexadecimal numerical system.

```
:03000000020100FA1001000075813F
7590FFB29012010D80F97A1479D40
90110003278589EAF3698E8EB25B
A585FEA2569AD96E6D8FED9FAD
AF6DD00000001FF255AFED589EA
F3698E8EB25BA585FEA2569AD96
DAC59700D00000278E6D8FED9FA
DAF6DD00000001FF255AFED8FED
9FADAF6DD000F7590FFB29013278
E6D8FED9FADAF6DD00000001FF2
55AFED589EAF3698E8EB25BA585
FEA2569AD96DAC59D9FADAF6D
D00000001FF255AFED8FED9FADA
F6DD000F7590FFB29013278E6D82
78E6D8FED9FA589EAF3698E8EB2
5BA585FEA2569AD96DAF6DD000
00001FF2DAF6DD00000001FF255A
ADAF6DD00000001FF255AFED8FE
D9FA
```

In case other software for program writing in assembly language is used, special software for compiling the program must be installed and used as follows: set up the compiler, open the document with .asm extension and compile. The result is the same- a new document with .hex extension. The only problem you have now is that it is stored in your PC.

PROGRAMMING A MICROCONTROLLER

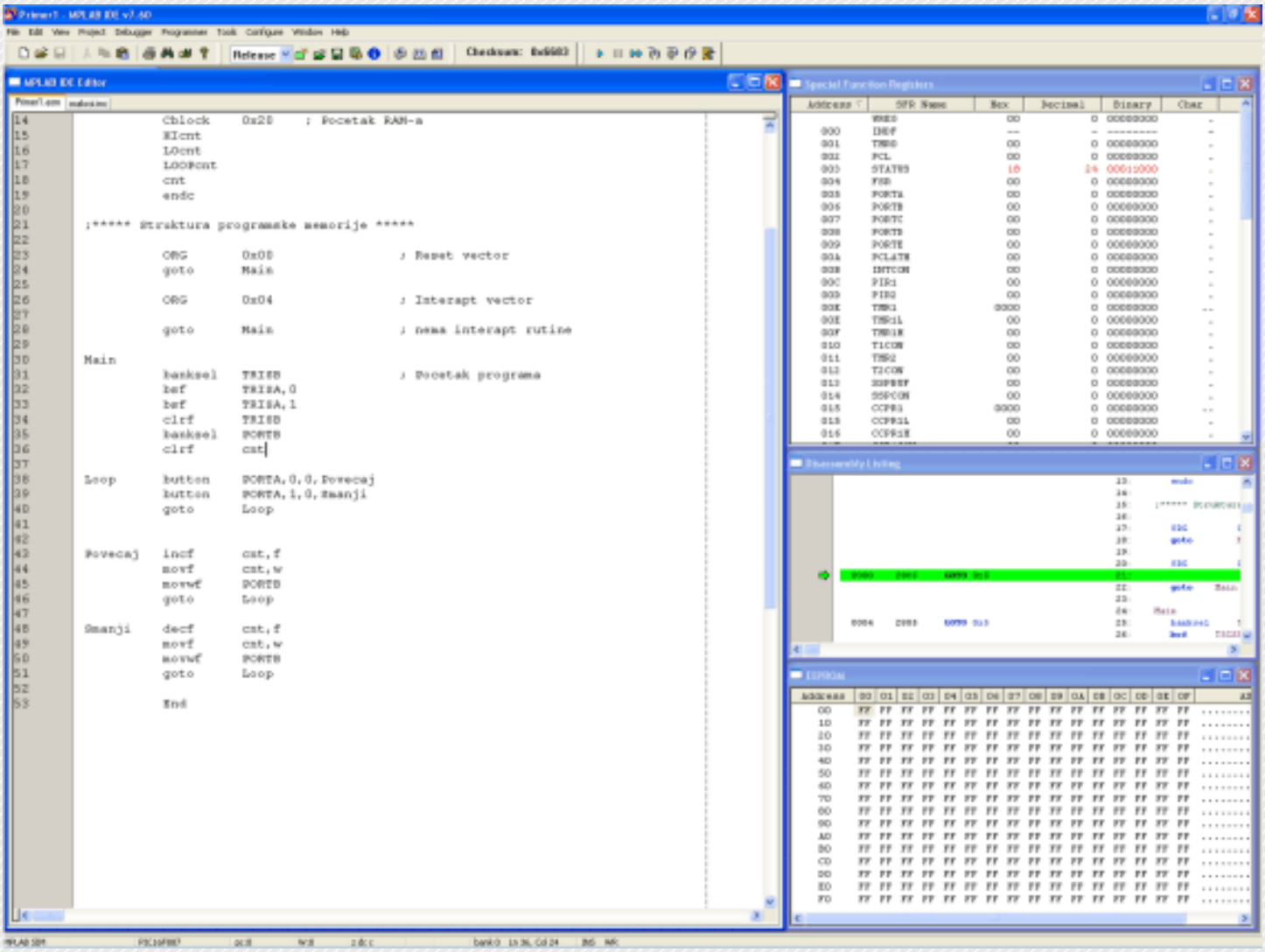
To enable "hex code" transmission to the microcontroller it is necessary to provide a cable for serial communication and a special device called programmer with appropriate software. There are several ways to do it.

A lot of programs and electronic circuits having this purpose can be found on the Internet. Do as follows: open hex code document, set a few parameters and click the icon for compiling. After a while, a sequence of zeros and ones is to be programmed into the microcontroller through the serial connection cable and programmer hardware. There is nothing else to be done except for placing the programmed chip into the target device. In case it is necessary to make some changes in the program, the previous procedure may be repeated an unlimited number of times.

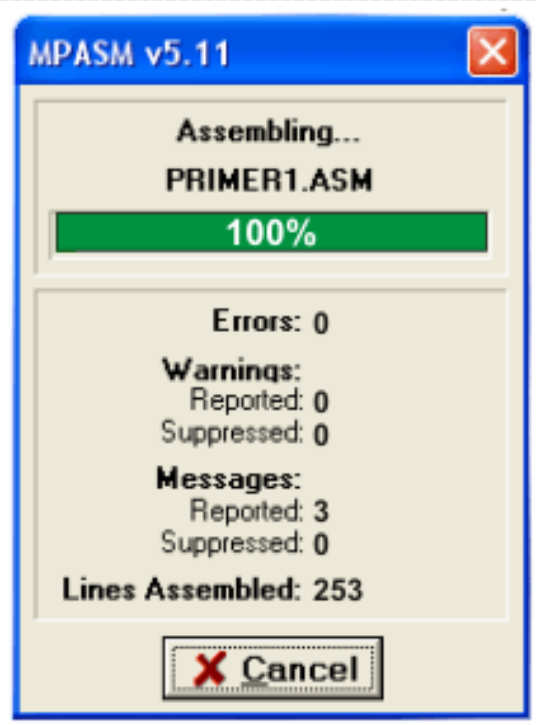
Is this a happy ending?

This section briefly describes the use of MPLAB and programmer software developed by Mikroelektronika. Everything is very simple...

You have already installed MPLAB, havenâ€™t you? Open a new project and a new document with extension.asm.



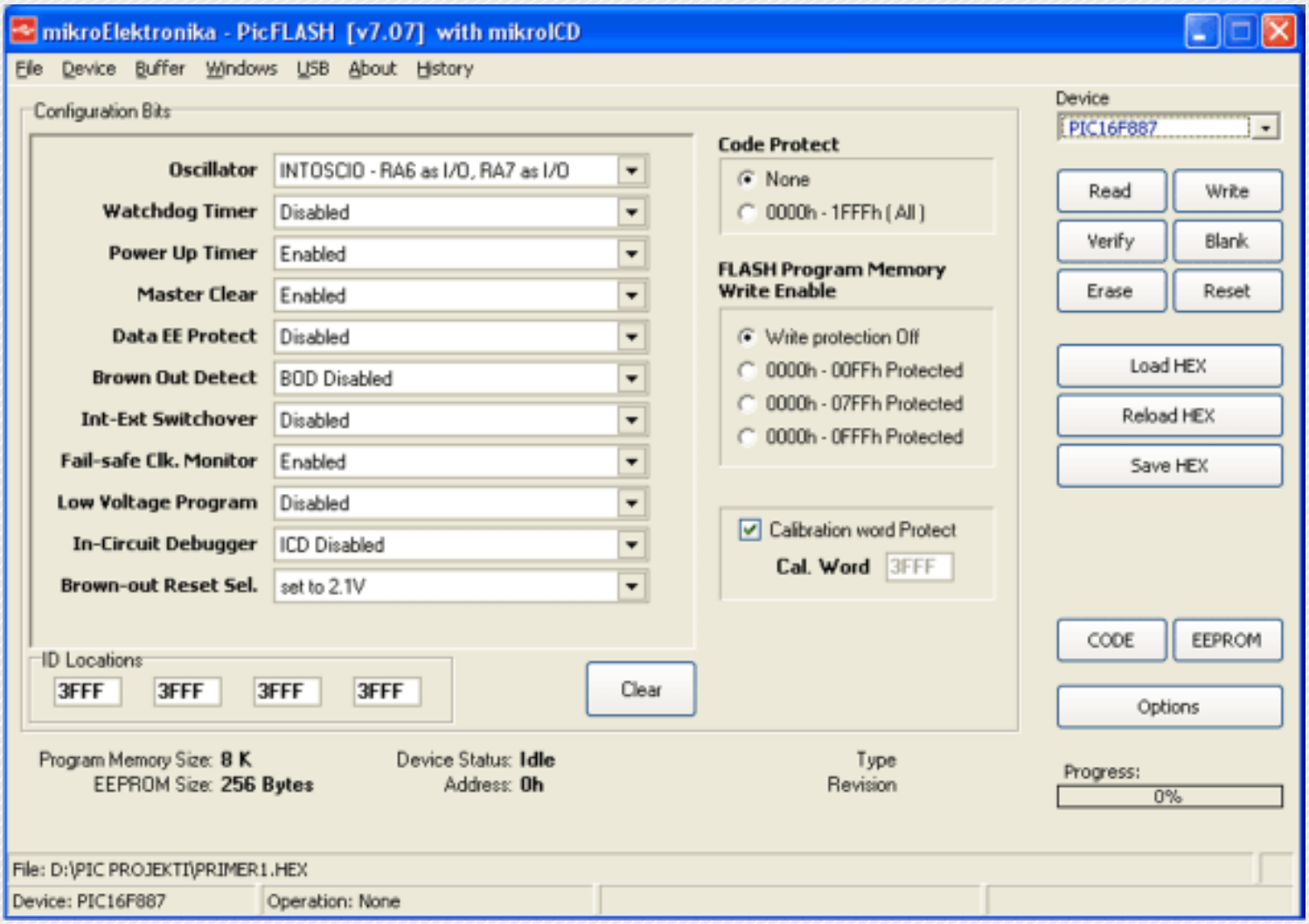
OK. You have written a program and tested it with the simulator. The program did not reports any error during the compiling process? It seems that everything is under control...



The program is written and successfully compiled. All that's left is to dump the program to the microcontroller. For this purpose it is necessary to have software that takes the written and compiled program and passes it into the microcontroller (PIC Flash for example). Start up this program...

The settings are simple and there is no need for additional explanations (the type of the microcontroller, frequency and clock oscillator etc.).

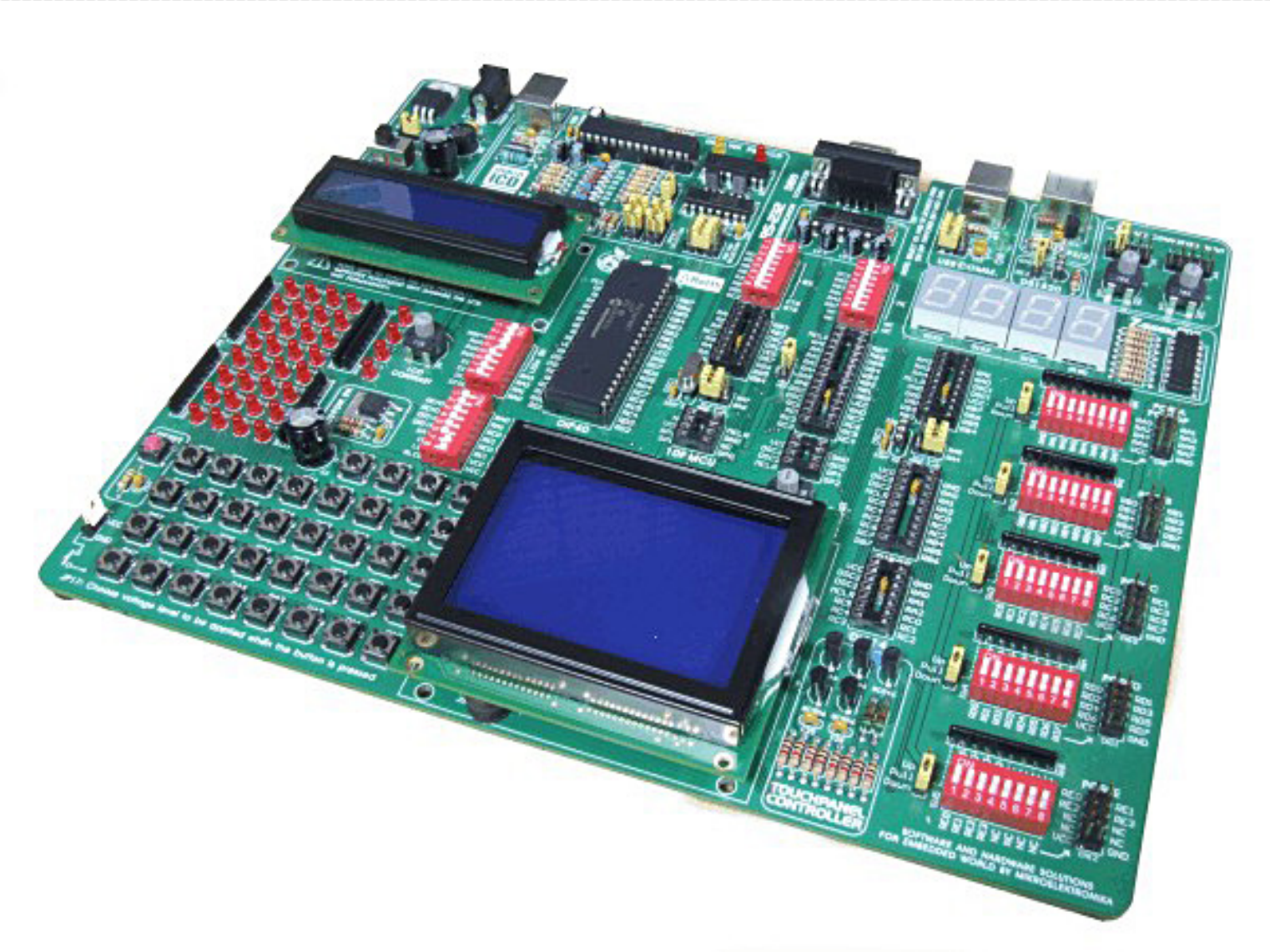
- Connect the PC and programmer via a USB cable;
- Load the HEX code using command: File -> Load HEX; and
- Click the "Write" push-button and wait...



That's it! The microcontroller is programmed and everything is ready for operation. If you are not satisfied, make some changes in the program and repeat the procedure. Until when? Until you feel satisfied...

Development systems

A device, which in testing program phase, can simulate any environment is called a development system. Apart from the programmer, the power supply unit and the microcontrollerâ€™s socket, the development system contains elements for input pin activation and output pin monitoring. The simplest version has every pin connected to one push-button and one LED as well. A high quality version has LED displays, LCD displays, temperature sensors and all other elements which the target device can be supplied with. These peripherals could be connected to the MCU via miniature jumpers. In this way, the whole program may be tested in practice, during its development stage, because the microcontroller does not know, or care, whether its input is activated by a push-button or a sensor built in a real machine.



Development system EasyPIC5

Previous Chapter | Table of Contents